```
 --            ---      ---       ----        ----
|  |\         /  /\    /  /\     /    |\     /    |\
|  ||        /  / /   /__/ /    /     ||    /     ||
|  ||       /  / /    \__\/    /  /|  ||   /  /|  ||
|  ||      /  / /   ___       /  / |  ||  /  / |  ||
|  ||     /  / /   /  /\     /  / /|  || /  / /|  ||
|  ||    /  / /   /  / /    /  / / |  ||/  / / |  ||
|  ||   /  / /   /  / /    /  / /  |  |/  / /  |  ||
|  ||  /  / /   /  / /    /  / /   |     / /   |  ||
|  || /  / /   /  / /    /  / /    |    / /    |  ||
|  ||/  / /   /  / /    /  / /     |   / /     |  ||
|  |/  / /   /  / /    /  / /      |__/ /      |  ||
|     / /   /  / /    /  / /       \__\/       |  ||
|    / /   /  / /    /  / /                    |  ||
|   / /   /  / /    /  / /                     |  ||
|__/ /   /__/ /    /__/ /                      |__||
\__\/    \__\/     \__\/                       \__\|
```

用户手册

作者： Bram Moolenaar
翻译：ZhaoRuFei[1]
版本： 603.0[2]

[1]slimzhao@hotmail.com
[2]从该版本开始，该中文文档的主版本将与软件的版本保持一致

译者前言

不要试图从本手册中去获取什么知识，使用Vim更多的是一种技能而不是一种知识，Vim的学习需要的更多的不是头脑而是双手，经常按书中的指示进行示例性的操作，在学习Vim众多精致的技巧时，不要贪图一下子全都掌握，最好是看一条技巧后，马上在编辑器上进行操作，这样在以后实际的编辑操作时你的手指就会建立一种自然的反应而不是由头脑来搜索该使用哪一条操作技巧．　建议读者不动手来不读书．如果手边没有一个合适的Vim编辑器环境可供操练，那么建议读者还是不要在这里浪费时间．

如果读者是在气温比较低的条件下阅读此书从而增加了你动手的惰性时，也请不要浪费时间，这会严重影响学习的效果．

虽然本书鼓励读者多动手，但也绝非说一点不要动脑，相反，Vim中多达几百个的命名与操作方式有它自己的规律可循，在你的手指能对要完成的编辑任务条件反射之前，最好还是由头脑做一点辅助．经常总结自己最经常进行的操作．为这些操作找出最简练的办法来，在每学习一条新的操作之前与自己以前的编辑经验比较一下，找出节省你敲击键盘次数的捷径来．　是提升Vim经验值的不二法门．

其实，Vim与其它编辑器一个很大的区别在于，它可以完成复杂的编辑与格式化功能．在这些领域还少有软件能与它分庭抗礼，但是，与所有的灵活性的代价一样，你需要用自己的双手来实现它．　这在事实上造成了用户在使用Vim过程中的几个自然阶段．

一开始是notepad，word，edit垄断你的大脑，这些东西根深蒂固，挥之不去．　Vim 的使用对你而言是一场噩梦，它降低而不是提高了你的工作效率．对三种工作模式的不解甚至使你认为它是一个充满BUG或者至少是一个古怪的与当今友好用户界面设计严重脱节的软件．事实上，这些起初看起来古怪的特性是Vim(或者是vi)的作者和它的用户们在自己漫长的文字编辑和程序设计生涯中总结出来的最快速最实在的操作，在几乎等于计算机本身历史的成长期中，历经无数严厉苛刻的计算机用户的批评与检验，无用的特性或糟糕的设计在Vim用户群面前根本就没有生存的余地．Vim细心而谨慎的作者们也不允许自己精心设计的软件里有这样的东西．

第二个阶段你开始熟悉一些基本的操作，这些操作足以应付你日常的工作，你使用这些操作时根本就不假思索．　但这些阶段你仍然很少去碰Vim那晦涩的在线帮助文档．　它在你心里只是notepad，edit一个勉强合格的替代品．

第三个阶段，精益求精的你不满足于无休无止的简单操作，冗长而乏味，有没有更好的办法可以驭繁于简？　于是，从UNIX参考手册上，从同事口中，你渐渐叩开:help xxx的大门．开始探索里面充满魔力的咒语．从杂耍般的带有表演性质的技巧开始，这些技巧令人眩目但少有实用性．　不过这却是你拥有魔力的第一步．接下来，你开始认识到这些咒语背后的真经，开始偷偷修改一些奇怪的符号，于是，奇迹产生了，魔力不但仍然有效，而且真实地作用于你现实中的文字编辑生活．　你

在第二阶段由于熟练操作而尘封已久的大脑突然开始运作. 但这个过程并非是达到某个临界状态后的一路坦途, 不断的挫折, 新的挑战, 看似Mission Impossible的任务. 永远伴随着任何一个人的任何一个学习过程. 这是你使用Vim的最后一个阶段, 也是最漫长最有挑战性同时也充满无数奇趣的阶段. 这个阶段里你开始定制一些希奇古怪的颜色. 开始以敲入i18n来输入internationalization, 开始让Vim替你纠正经常把the 误敲成teh的毛病, 开始让Vim与系统里各种精悍而强大的兄弟工具进行合作, 开始写越来越长的script, 每一次的文本编辑体验都妙趣横生高潮跌起. 你的头脑因为要用Vim完成高效的编辑而高度紧张. 你开始在Vim邮件列表里提一些确实是问题的问题. 也开始发现你在Vim里做了以前在SHELL里做的几乎一切事. 事实上你已经成了一个无可救药的Vim骨灰级玩家.

以上就是一个Vim用户的精神之旅.

本文档仍在进一步完善中, 原因有三, 一为技术本身, 译者虽在Vim的大量命令、选项中饱经浸染,但不敢妄言说了解Vim的方方面面, 二为翻译, 有些译法有些术语欠妥, 我自己换个时间看感觉就不一样, 此一时也, 彼一时也; 三为用LaTeX制作期间, 又因LaTeX中对一些符号的特殊处理引入的错误. 以我一己之力要字斟句酌实在难为, 犹豫再三, 还是拿出来献丑, 把它放在众人的显微镜下,任何错误、翻译术语的建议、错别字可以email 给slimzhao@hotmail.com

下面是手册中关于这份文档的版权, 我举双手双脚赞成.

附录是由我翻译的几篇有关Vim的文章.

slimzhao@hotmail.com
2004/06/08

关于0.2版

0.2版与0.1版有如下改动:

段首的缩进改为2个汉字的宽度，看起来更舒服一些(或更不舒服一些).

修改了书签在Acrobat中显示出现乱码的问题. 该问题在Acrobat Reader的版本5和版本6中均有不同程度的表现.

第3章第4小节的第一行的

===============
此时使用命令”与
===============

中的命令应为%

感谢钱震(qzhen@flotu.org)提供修改建议和关于gbk2uni.exe的信息

感谢CTex的所有制作人员和论坛的热心网友们

关于6.3.0版

1. 为什么一下跳到6.3.0了?

我最新得到的VIM是6.3,它的文档与我最初翻译所基于的6.1有少许不同.这使我萌生了将文档更新至最新版本的想法. 那么,为何不让中文版与软件保持同步呢?

2. 6.3.0与0.2版相比有如下改动:

(A) 0.1版与0.2版居然都缺了usr_21.txt, usr_22.txt, usr_29.txt, usr_45.txt这4个文件,这4个文件与其它文件是一并翻译完毕的,打包处理时竟漏掉了.

(B) 将文档中所有出现的示例性的代码,shell命令,屏显样例,都重新格式化,如下:

ex command
```
:au BufRead *.tip setf tip
```

看起来更醒目一些.

我把示例性的代码分为7类. 一类是上面显示的这种,Vim中的命令行模式,标题是ex command,另外6种分别是:

normal mode command
Vim的normal模式命令,特点是没有冒号打头. 如

normal mode command
```
gg=G
```

shell command
多数情况下为unix类系统下的shell命令,也可能是windows OS的DOS BOX中的命令. 如:

shell command
```
env LANG=de_DE.ISO_8859-1  vim
```

Display
对应于一个命令执行完之后的屏幕显示结果,或操作的某种中间状态. 如:

Display

```
:!make | &tee /tmp/vim215953.err
gcc -g -Wall -o prog main.c sub.c
main.c: In function 'main':
main.c:6: too many arguments to function 'do_sub'
main.c: At top level:
main.c:10: parse error before '}'
make: *** [prog] Error 1

2 returned
"main.c" 11L, 111C
(3 of 6): too many arguments to function 'do_sub'
Hit ENTER or type command to continue
```

List
显示的是一系列列表值，如：

List

r	在Insert模式下按下回车时插入一个星号
o	在Normal模式下按"o"或"O"时插入一个星号
c	根据'textwidth'的设置自动为注释断行

code，如：

code

```
while i < b {
        if a {
                b = c;
        }
}
```

url，如：

URL

http://www.cl.cam.ac.uk/~mgk25/download/ucs-fonts.tar.gz

(C) 把混杂在汉字中的英文的默认字体改成了cmtt，即computer modern typewriter，个人品味，比cmr(computer modern roman)好看. 除了个别的情况，如显示Latex系统中的标志性建筑LaTeX时用的还是cmr，如果用cmtt就会是这样：LaTeX. 中间那个A显得很萎缩.

(D) 文档内也加了超级链接，但只限于这份中文用户手册内部，如果指向参数手册的文档，则没有超级链接，目前这份中文文档只限于用户手册.

另外指向外部的URL也能打开你的默认浏览器了. 指向email地址的也会打开你的默认mail程序.

(E) 所有的双引号不再显示为‘‘...’’(LaTeX风格)，也不是汉字中的“...”，而显示为最朴素的"..."，同样，以前显示为’的字符，在该版本中显示为'，这样的目的一是为了与原文档的风格更接近，再者也更悦目一些． 而正反斜杠的显示为/hello \hi，粗粗壮壮，看上去很厚道。不象这样\hi。上下都长出一截，看起来突兀．

(F) 对一些原文档中特殊显示的元素以同样的风格高亮起来，如CTRL-N，<Esc>

(G) 修改了一些BUG，对不那么顺畅的词句重新修遣一番． 对原来译文中留下的一些不甚了了的技术问题仔细对照文档，试验，修改，添加...

(H) 加了背景色，PDF不比HTML，你打开HTML源文件加一个bgcolor就可以改改背景色．这是个小问题，但是PDF中显示汉字有一个笔画太淡的问题．这在各种PDF制作或Tex相关的论坛上也是一个热门话题．看起来对比度不够． 现在的这个背景能让汉字显得更清晰一些．另外． 本文档拟提供一个内嵌字体的伴生版．其中的字体不是宋体，是文鼎公司贡献给自由软件社区的PL简报宋(Public License，不是Piao Liang)． 这个字体比起windows自带的宋体字着墨要浓一些． 当然这也导致文件会比较大．

3． TODO:

似乎每解决一个问题，都会引起更多新的问题．比如，好不容易找到了一些叫upquote的LaTeX package，也能把原来的’变成现在的'了． 可是却发现普通文字中的这个符号与示例代码中的显示不一样,后者显示出来的更粗壮一些.通过fancyvrb把示例代码弄得更好看一些了，可是却发现那个圈起来的框框有时候罩不住里面的文字---里面的文字太长了． 这些虽是小问题，终究让人不爽．我不是LaTeX专家，这些鸡毛蒜皮的事可TMD费功夫了，虽然现在这个样子我还不是很满意，但总得把它锁定到一个状态．就这了．

usr_toc.txt Vim 6.3版.最后修改: 2003年08月18日

VIM用户手册--- 作者: Bram Moolenaar

翻译: slimzhao@hotmail.com

目录 *user-manual*

概览

起步

高效编辑

打造Vim

运转Vim

可以在下面的地址中找到以单个文件组织的可打印版的HTML或PDF格式用户手册:

http://vimdoc.sf.net

起步

请从头至尾细读本章，本章讲述Vim的基本命令.

高效编辑

此类主题可以独立阅读

[1]译：仅对Unix类系统有意义

调节Vim

让Vim如你所愿地工作

运转Vim

Vimming之前.

---

usr_01.txt Vim 6.3版.最后修改: 2004年05月01日

VIM用户手册--- 作者: Bram Moolenaar

关于本手册

本章介绍Vim的帮助系统. 本文将使你了解到Vim的帮助中解讲每个命令时的假设环境.

下一章: |usr_02.txt| 迈出第一步
目 录: |usr_toc.txt|

01.1 两套帮助

Vim的文档由两部分组成:

1. 用户手册面向问题, 由浅入深进行讲解. 可以象读一本书一样从头至尾进行学习.

2. 参考手册详述Vim方方面面的细节.

这些手册中用到的一些约定的记法可以在|notation|中找到.

跳转

两个管道符之间的超级链接可以使你直接跳转到对该主题的解释处. 或者是对相应的编辑任务的应对之计, 或者是对它的功能进行详尽的解释. 牢记下面的两个命令[1]:

List

```
CTRL-]  跳转到当前光标所在单词对应的主题
CTRL-O  回到前一个位置
```

[1]译: 不要误会这样的说法, Vim中超级链接的实现跟HTML中不一样, click here中的click here跟作为普通文本的"click here"是不一样的, 单击前者会跳转, 后者不会; 而Vim中两个管道符圈起一个词条作为一个帮助主题并不是说只有将光标置于此处按CTRL-]才可跳转, 它是说明性的, 如果普通文本中碰巧有一个单词跟某个帮助主题相同, 在它上面施以CTRL-]效果也完全一样

很多的链接都写在两个管道符中，象这个：|bars|. 一个选项，如'number'，或者是一个命令如":write"，或者任何其它的词都可以作为一个超级链接. 试一下移动光标到CTRL-]上按下CTRL-].

其它的帮助主题可以通过":help"命令来访问，请参考|help.txt|.

01.2 关于安装

手册中假设你已正确地安装了Vim. 如果你还没有，或者装了但不能运行(比如找不到文件或GUI菜单显示不出来)，请先阅读关于安装的手册：|usr_90.txt|.

not-compatible

手册中也假设你是在Vi兼容模式关闭的情况下使用Vim的. 对绝大多数命令来说是否是Vi兼容都没有问题，但有时这一点会变得很重要，比如对于多级撤消. 保证你进行正确设置的最简单办法就是复制一个样板vimrc文件. 在VIM内部复制的话你甚至无需知道它的具体位置，不过文件名因系统而异：Unix:

ex command
```
:!cp -i $VIMRUNTIME/vimrc_example.vim ~/.vimrc
```

MS-DOS，MS-Windows，OS/2:

ex command
```
:!copy $VIMRUNTIME/vimrc_example.vim $VIM/_vimrc
```

Amiga:

ex command
```
:!copy $VIMRUNTIME/vimrc_example.vim $VIM/.vimrc
```

如果同名文件已经存在你也许还想保留下来.

如果你现在启动Vim，'compatible'选项应该是关闭的. 下面的命令可以检查它的设置：

ex command
```
:set compatible?
```

如果结果是'nocompatible'就对了. 如果是"compatible"可就麻烦了. 你要找找看为什么这个选项还是这样的设置. 也许是上面的文件没找到. 下面的命令可以告诉你它的位置：

ex command
```
:scriptnames
```

如果你的配置文件没有在这个列表中，你需要检查一下它的位置和名字.如果在，那一定是别的什么地方把'compatible'选项给打开了.

详情请参考|vimrc|和|compatible-default|.

备注：本手册是关于以常规方式使用Vim. 还有一个叫"evim"(easy vim)的程序. 它也是Vim. 但是被改装成了类似于Notepad的风格. 它总是处于Insert模式，感觉很难受. 本手册不对此提供帮助，因为它太简单了，一看就会. 关于它的细节请参考|evim-keys|.

01.3 使用Vim教程

除了阅读文档(无聊!)你还可以用vimtutor来开始学习Vim的简单命令. 这是一个大概30 分钟的教程，它会教给你最常用的基本操作.

在Unix和MS-Windows上，如果Vim安装好了，你可以这样进入该教程：

shell command
```
vimtutor
```

它会复制一份教程文件，这样你可以在其中放心地练习，不用担心破坏了原来的内容.

本教程有几个译本. 要看看你的本国语是否已被翻译,可以在命令后加两个字符的语言代码试试，如对法语：

shell command
```
vimtutor fr
```

对于OpenVMS系统，如果VIM已正确安装，可以用下面的命令进入教程：

shell command
```
@VIM:vimtutor
```

在其它系统上，你可要费点小事：

1. 复制教程文件. 你可以在VIM中做(它知道文件的位置)：

shell command
```
vim -u NONE -c 'e $VIMRUNTIME/tutor/tutor' -c 'w! TUTORCOPY' -c 'q'
```

本，只需在

文件名后追加它的对应的两个字符的语言代码，比如对法语：

shell command
```
vim -u NONE -c 'e $VIMRUNTIME/tutor/tutor.fr' -c 'w! TUTORCOPY' -c 'q'
vim -u NONE -c "set nocp" TUTORCOPY
```

这两个参数会让Vim更happy一些.

3. 学完后把教程文件删掉del TUTORCOPY

01.4 版权 *manual-copyright*

frombook

本手册的部分内容来自Steve Oualline的≪Vi IMproved - Vim≫一书(由New Riders出版公司发行, ISBN:0735710015). "开放出版许可证"也同样适用于该书, 该书被本手册引用的部分也已作出修改(比如, 去掉了一些图片, 更新了一些Vim 6.0版相关的内容以及修改了一些错误). 没有|frombook|标签的地方可并不是说一定就不是来自该书.

多谢Steve Oualline和New Riders出版社制作了该书并以OPL的形式出版! 它对我写这份手册大有帮助. 不光是因为它提供了文字素材, 也决定了这份手册的风格和基调.

如果你通过出售该手册谋利的话, 我强烈建议你把部分收益捐助给乌干达的爱滋病患者. 请参考|iccf|.

下一章: |usr_02.txt| 迈出第一步
版 权: 请参考|manual-copyright| vim:tw=78:ts=8:ft=help:norl:

usr_02.txt Vim 6.3版.最后修改: 2003年05月04日

VIM用户手册--- 作者: Bram Moolenaar

Vim第一步

本章仅提供可以让你开始用Vim编辑文件的必要技巧. 所用的方法可能既不是最好的也不是最快的. 它只是让你有一个开端. 你最好花些时间去实际应用一下这些命令, 它们是进一步学习的基础.

下一章: |usr_03.txt| 移动
前一章: |usr_01.txt| 关于本手册
目　录: |usr_toc.txt|

02.1 首次运行Vim

命令:

shell command
```
gvim file.txt
```

可以启动Vim.

在UNIX下你可以直接在命令行上键入该命令, 但如果你用的是Microsoft Windows, 就需要在一个MS-DOS命令行窗口中键入.

上面的命令使Vim开始编辑一个名为file.txt的文件. 因为这是一个新文件, 所以你会看到一个空的窗口. 屏幕上看起来大致是这样:

Display
```
+---------------------------------------+
|#                                      |
|~                                      |
|~                                      |
|~                                      |
|~                                      |
|"file.txt" [New file]                  |
+---------------------------------------+
```

("#" 代表当前光标位置.)

上波浪线(~)表示所在行并不是文件内容的一部分. 换句话说, Vim将文件之外的部分显示为波浪线. 在窗口的底部, 一个消息行显示说当前正在编辑的文件叫file.txt, 它是一个新文件. 显示的消息总是临时性的, 系统中显示的其它消息会覆盖掉前面的消息.

VIM命令

gvim命令使编辑器打开一个新的窗口进行编辑. 如果你用的是命令:

shell command
```
vim file.txt
```

就会在当前的命令行窗口中打开编辑程序. 或者说, 你在运行xterm的话, 编辑用的窗口就是你当前的xterm窗口. 如果你用的是Microsoft Windows下的MS-DOS命令行窗口, 编辑器就在该命令行窗口中打开. 两种情况下窗口中显示的内容都是一样的, 但是用gvim 的话可以使用额外的功能, 如菜单等.

02.2 插入文本

Vim编辑器是一个模式编辑器. 这意味着在不同状态下编辑器有不同的行为模式. 两个基本的模式是Normal模式和插入模式. 在Normal模式下你键入的每一个字符都被视为一个命令. 而在Insert模式下键入的字符都作为实际要输入的文本内容.

刚启动时Vim工作于Normal模式. 要进入Insert模式你需要使用"i"命令(i 意为Insert). 接下来就可以直接输入了. 别怕出错, 错了还可以修改. 比如下面这首程序员的打油诗:

Display
```
iA very intelligent turtle
Found programming UNIX a hurdle
```

"turtle"之后你按下回车键另起一行. 最后按下<Esc>键退出Insert模式回到Normal模式.现在你的Vim窗口中有了这样的两行内容:

Display
```
+----------------------------------------+
|A very intelligent turtle               |
|Found programming UNIX a hurdle         |
|~                                       |
|~                                       |
|                                        |
+----------------------------------------+
```

现在是什么模式?

要知道你现在所处的工作模式是什么，打开显示模式的开关:

ex command
```
:set showmode
```

你会看到按下冒号键之后当前光标跑到窗口的最后一行去了. 那是使用冒号命令的地方(顾名思义，冒号命令就是总是以冒号打头的命令). 最后按下回车键结束整个命令(所有的冒号命令都以这种方式表明命令的结束).

现在，如果你键入了"i"命令Vim就会在窗口底部显示--INSERT--. 这表明你目前处于Insert模式.

Display
```
+----------------------------------------+
|A very intelligent turtle               |
|Found programming UNIX a hurdle         |
|~                                       |
|~                                       |
|-- INSERT --                            |
+----------------------------------------+
```

如果按下<Esc>键返回到Normal模式刚才显示出来的模式"--INSERT--"就会消失[1]

模式之灾

Vim新手最头痛的问题就是模式---经常忘记自己置身于何种模式，或者不经意敲了哪个字符就切换到别的模式去了. 不管你当前所处的模式是什么，按下<Esc>都会让你回到Normal模式(即使已经在Normal模式

[1]译: Normal模式并不会显示--NORMAL--，作为默认的工作模式它不显示任何字串

下）． 有时需要按两次<Esc>，如果Vim以一声蜂鸣回答你，那说明你已经是在Normal模式了[1]

02.3 移动光标

回到Normal模式后，你就可以用下面的命令来移动光标：

List

```
h 左
j 下
k 上
l 右
```

人们一开始会认为这些字符是随意选取的． 毕竟有谁拿l来代表right呢？但事实上，这些字符都是精心挑选的： 在编辑器中移动光标是十分常用的操作，这些字符在键盘上都分布在你右手周围．这样的安排可以使你最快最方便地使用它们(尤其是是对那些用十个手指而不是二指禅用户而言)

> 备注： 同时你还可以用箭头键来移动光标．不过这样做实际上会大大降低你的效率．因为用这些键你需要不停地在字母区和箭头键之间频繁转换．想象一下要是你在一小时内这样做一百次会占用你多少时间？ 另外，并不是每个键盘上都安排有箭头键，或者都把它们放在最常见的位置；所以使用hjkl还是大有好处．

记住这些命令的一个办法是通过它们在键盘上的布局：h在左边，l在右边，j指向下面．

	k	
h		l
	j	

但学习这些命令的最好办法不是使用什么记忆法，而是练习．你可以用"i"命令来在Insert 模式下输入一些内容，然后用hjkl命令将光标移到别处再插入另外的内容，不要忘了要用<Esc>来回到Normal模式．|vimtutor|也是学习这些命令的一个好去处．

对于日本的用户来说，Hiroshi Iwatani有如下建议：

[1]译：在google的新闻组上还有人写了一首诗来表达这种困扰，可惜我再没找到它了

Display
```
                Komsomolsk
                    ^
                    |
   Huan Ho      <--- --->  Los Angeles
(Yellow river)      |
                    v
                  Java (the island, not the programming language)
```

02.4 删除字符

要删除一个字符，只需要将光标移到该字符上按下"x"．（这是在追忆古老的打字机时代，在打字机上删除字符就是用xxxx来覆盖它）把光标移到上面例子中的第一行，键入xxxxxxx(7个x) 来删除"A very "． 结果如下：

Display
```
+---------------------------------------+
|intelligent turtle                     |
|Found programming UNIX a hurdle        |
|~                                      |
|~                                      |
|                                       |
+---------------------------------------+
```

现在你可以键入其它内容了，比如：

Display
```
iA young <Esc>
```

首先键入的命令是i(进入Insert模式)，接着插入"A young"，然后退出Insert模式(最后的<Esc>)． 结果是：

Display
```
+---------------------------------------+
|A young intelligent turtle             |
|Found programming UNIX a hurdle        |
|~                                      |
|~                                      |
|                                       |
+---------------------------------------+
```

删除一行

删除一整行内容使用"dd"命令． 删除后下面的行会移上来填补空缺：

Display
```
+----------------------------------------+
|Found programming UNIX a hurdle         |
|~                                       |
|~                                       |
|~                                       |
|                                        |
+----------------------------------------+
```

删除换行符

在Vim中你可以把两行合并为一行，也就是说两行之间的换行符被删除了： 命令是"J". 比如下面的两行：

Display
```
A young intelligent
turtle
```

将光标移到第一行上然后按"J"：

Display
```
A young intelligent turtle
```

如果你误删了过多的内容. 显然你可以再输入一遍，但是命令"u" 更简便，它可以撤消上一次的操作[1].. 实际看一下它的效果，用"dd"命令来删除前面例子中的第一行内容，"u"命令会把它找回来. 另一个例子：将光标移到第一行的A上：

Display
```
A young intelligent turtle
```

现在用命令xxxxxxx来删除"A young". 结果如下：

Display
```
 intelligent turtle
```

键入"u"来撤消最后的一次删除. 最后被删除的是字符g，所以撤消操作恢复了这个字符：

Display
```
g intelligent turtle
```

[1]译：不要误解为它只能删除最后一次的操作，它也可以删除上上次，上上上上...次的操作

下一个u命令将恢复倒数第二次被删除的字符:

Display
```
ng intelligent turtle
```

再下一次是字符u，如此如此:

Display
```
ung intelligent turtle
oung intelligent turtle
young intelligent turtle
 young intelligent turtle
A young intelligent turtle
```

备注: 如果你按下"u"两次结果是两次找回了同样的字符，那说明你的Vim配置成Vi兼容模式了. 在|not-compatible|可以找到这一问题的对策. 这个例子假设你的Vim使用的是Vim的方法. 如果你更喜欢老的Vi编辑器的做法，你就要留心两者在这方面的细微差别.

重做

如果你撤消了多次，你还可以用CTRL-R(重做)来反转撤消的动作. 换句话说，它是对撤消的撤消. 实际按两次CTRL-R试试它的效果，字符A和它后面的空格又出现了:

Display
```
young intelligent turtle
```

撤消命令还有另一种形式，"U"命令，它一次撤消对一行的全部操作. 第二次使用该命令则会撤消前一个"U"的操作.

Display
```
A very intelligent turtle
  xxxx                          删除very

A intelligent turtle
            xxxxxx              删除turtle

A intelligent
                                用"U"恢复该行
A very intelligent turtle
                                用"u"撤消"U"
A intelligent
```

"U"命令本身也造成了一次改变，这种改变同样可以用"u"命令和CTRL-R来撤消和重做．看起来这很容易把人搞糊涂，不过别担心，用"u"和CTRL-R你可以找回任何一个操作状态．

02.6 其它编辑命令

Vim有一大堆命令来改变文本． 请参考|Q_in|和下面的内容，这里仅列出一些最常用的．

追加

"i"命令可以在当前光标之前插入文本． 但如果你想在当前行的末尾添加一些内容时怎么办呢? 你必需在光标之后插入文本． 答案是用"a"命令来代替"i"．

比如，要把

Display
```
and that's not saying much for the turtle.
```

改变为

Display
```
and that's not saying much for the turtle!!!
```

把光标移到行尾的句点上，然后用"x"来删除这个点号．现在光标被置于行尾turtle的e上了． 键入命令：

normal mode command
```
a!!!<Esc>
```

来在e的后面追加三个感叹号：

Display
```
and that's not saying much for the turtle!!!
```

另起一行

"o"命令可以在当前行的下面另起一行，并使当前模式转为Insert模式． 这样你可以在该命令之后直接输入内容．假设光标位于下面两行中第一行的某处：

Display
```
A very intelligent turtle
Found programming UNIX a hurdle
```

现在键入命令"o"并输入下面的内容:

normal mode command
```
oThat liked using Vim<Esc>
```

结果将是:

Display
```
A very intelligent turtle
That liked using Vim
Found programming UNIX a hurdle
```

"O"命令(注意是大写的字母O)将在当前行的上面另起一行.

使用命令计数

假设你要向上移动9行. 这可以用"kkkkkkkkk"或"9k"来完成. 事实上, 很多命令都可以接受一个数字作为重复执行同一命令的次数. 比如刚才的例子, 要在行尾追加三个感叹号, 当时用的命令是"a!!!<Esc>". 另一个办法是用"3a!<Esc>"命令. 3说明该命令将被重复执行3次. 同样, 删除3个字符可以用"3x". 指定的数字要紧挨在它所要修饰的命令前面.

02.7 退出

要退出Vim, 用命令"ZZ". 该命令保存当前文件并退出Vim.

> 备注: Vim不会象其它的编辑器那样, 自动为被编辑的文件生成一个备份. 如果你用"ZZ", Vim就会提交你对该文件所做出的修改. 并且无法撤消. 当然你也可以配置你的Vim让它也可以自动生成一个备份文件. 请参考|07.4|.

放弃编辑

有时你会在做了一连串修改之后突然意识到最好是放弃所有的修改重新来过. 别担心. Vim中有一个命令可以丢弃所有的修改并退出:

ex command
```
:q!
```

别忘了在命令之后加回车.

对于喜欢把事情弄出个究竟的人来说，这个命令由3部分组成： 冒号(:)，用以进入冒号命令行模式；q命令，告诉编辑器退出；最后是强制命令执行的修饰符(.!) [1] [2].

这里强制命令执行的修饰符是必需的，因为Vim对隐含地放弃所有修改感到不妥.如果你只是用":q"来退出，Vim会显示下面的错误消息并且拒绝不负责任地退出：

Display
```
E37: No write since last change (use ! to override)
```

指定了强制执行的修饰符等于你告诉Vim，"我知道也许我这样做很蠢，但是我已经长大了我知道我在做什么你就让我蠢一次吧".

如果你在放弃所有修改后还想以该文件的初始内容作为开始继续编辑，还可以用":e!"命令放弃所有修改并重新载入该文件的原始内容.

02.8 求助

你想做的任何操作都可以在Vim的帮助文件里找到答案，别怕问问题！命令

ex command
```
:help
```

会带你到帮助文件的起始点． 如果你的键盘上有一个<F1>键的话你也可以直接按<F1>.

如果你没有指定一个具体的帮助主题，":help"命令会显示上面提到的帮助文件的起点． Vim的作者聪明地(也许是懒惰地)设计了它的帮助系统：帮助窗口也是一个普通的编辑窗口． 你可以使用跟编辑其它文件时一样的命令来操作帮助文件．比如用hljk 移动光标．

退出帮助窗口也跟退出其它文件编辑窗口一样，使用"ZZ"即可．这只会关闭帮助窗口，而不是退出Vim.

浏览帮助文件时，你会注意到有一些内容用两个小栅栏围了起来(比如|help|)． 这表明此处是一个超链接． 如果你把光标置于两个小栅栏之间的任何位置然后按下CTRL-](跳转到一个标签的命令)，帮助系统就会带你到那个指定的主题．(因为一些此处不便讨论的原因，在Vim的术语中这种超链接叫标签． 所以CTRL-]可以跳转到当前光标之下的那个word所对应的链接中[3].

[1]译：注意千万不要以为Vim的基本命令可以象这样任意组合成一个新的命令

[2]译：规律:!表示强制命令执行，强制执行的结果要视具体的命令而定，如w!是覆盖已经存在的文件

[3] 译：基本上，这与HTML语言中的超链接在概念和功能上是一样的,只不过HTML语言

几次跳转之后，你可能想回到原来的某个地方，CTRL-T(弹出标签)可以回到前一个位置．用命令CTRL-O(跳转到较早的位置)也可以．

帮助窗口的开始有一个关于*help.txt*的说明．在星号"*"之间的字符被帮助系统定义为一个标签的目的地(超链接的目的地)．

参考|29.1|可以了解更多关于标签使用的细节．

要查看关于某个特殊主题的帮助，使用下面的命令形式：

ex command
```
:help {subject}
```

比如要得到关于"x"命令的帮助，就可以使用：

ex command
```
:help x
```

要查找关于如何删除的内容，使用命令：

ex command
```
:help deleting
```

要得到所有Vim命令的索引，使用命令：

ex command
```
:help index
```

如果你要得到关于某个控制字符的帮助(比如，CTRL-A)，你需要用前缀"CTRL-"来代表控制键：

ex command
```
:help CTRL-A
```

Vim编辑器有很多模式[1]．默认情况下帮助系统显示的是Normal模式下某个命令的帮助．比如，下面的命令显示Normal模式下CTRL-H命令的帮助：

ex command
```
:help CTRL-H
```

要查找其它模式下的帮助，使用一个模式前缀．如果你想要看的是Insert模式下某个命令的帮助，使用"i_"前缀．对于CTRL-H来说是这样：

中对超链接的定义是
HTML
而在Vim中对超链接目的地的描述是通过tag文件中的一个条目，这个条目用一种搜索或定位命令的形式告诉Vim目的地在哪里

[1]译：不光是前面提到的Insert模式和Normal模式

ex command
```
:help i_CTRL-H
```

启动Vim编辑器时，你可以使用一些命令行参数． 这些参数都以-开始．比如说要查找-t参数的功能，使用命令：

ex command
```
:help -t
```

Vim编辑器也有众多的选项来让用户自己进行定制．如果你想得到关于某个选项的帮助，你需要把它用单引号括起来． 比如要找'number'选项，就可以用命令：

ex command
```
:help 'number'
```

关于各种模式都要用哪些前缀可以在|help-context|中找到．

特殊键用尖括号中一个简单的描述性名字表示． 比如要查找Insert模式下的上箭头键的功能，可以用：

ex command
```
:help i_<Up>
```

如果你看到了象下面这样的错误信息还是不明究竟：

Display
```
E37: No write since last change (use ! to override)
```

你可以把它的错误ID号作为一个帮助主题来得到更进一步的信息：

ex command
```
:help E37
```

下一章：	|usr_03.txt| 移动
版　权：	请参考|manual-copyright| vim:tw=78:ts=8:ft=help:norl:

usr_03.txt Vim 6.3版.最后修改: 2004年01月17日

VIM用户手册--- 作者: Bram Moolenaar

移动

在你插入或删除文本之前，光标当然要先移动到正确的位置上，Vim有众多的命令来移动光标. 本章将介绍这些命令中最重要的一些. 此外，你可以在|Q_lr|找到这些命令的完整列表.

下一章: |usr_04.txt| 小幅改动
前一章: |usr_02.txt| 迈出第一步
目 录: |usr_toc.txt|

03.1 以Word为单位的移动

使用"w"命令可以将光标向前移动一个word. 象大多数其它的Vim命令一样，你可以在"w"前面指定一个数字前缀以向前移动指定个数的word. 比如"3w"将光标向前移动3个words. 请看下面的命令示意:

```
Display
This is a line with example text
 --->-->->----------------->
  w  w  w    3w
```

注意如果当前光标已经在一个word的首字符上时"w"命令还是会将光标移动到下一个word的首字符上. "b"[1]命令则将光标向后移动到前一个word的首字符上:

[1]译注: 助记: backward

```
                               Display
        This is a line with example text
        <----<--<-<---------<---
           b    b b    2b      b
```

同样，"e"[1]命令会将光标移动到下一个word的最后一个字符[2].象"w"有一个反方向的命令"b"与之对应一样，"e"命令有"ge"，它将光标移动到前一个word的最后一个字符上[3]:

```
                               Display
        This is a line with example text
           <-   <--- ----->   ---->
           ge     ge     e       e
```

如果光标已经位于当前行的最后一个word，则"w"会移动到下一行的第一个word上去． 所以使用"w"就可以在整个文本段中移动，速度要比"l" 快多了． "b"也一样，只是方向相反．

有一些被认为是non-word的特殊字符，比如"."，"-"或")"充当了word边界的作用． 要改变Vim对word边界的定义，请查看'iskeyword'选项.还可以以空白为分界的WORDs为单位进行移动．这种WORD与通常意义上的word的边界不同.所以此处用了大写的WORD来区分于word． 它的移动命令也是相应字母的大写形式，如下所示:

```
                               Display
        ge      b          w                             e
        <-     <-         --->                          --->
 This is-a line, with special/separated/words (and some more).
    <----- <-----         -------------------->         ----->
      gE      B                    W                       E
```

混合使用这种不同大小写的命令，你可以更快地在文本中前后移动．

03.2 移动到行首或行尾

"$"命令将光标移动到当前行行尾．如果你的键盘上有一个<End>键，它的作用也一样．

"^"命令将光标移动到当前行的第一个非空白字符上[4]． "0"命令则总

[1]译注： 助记:end of word

[2]译注： 与命令"w"不同，如果当前光标在当前word上的位置不是最后一个字符，则"e"命令会把光标移动到当前word的最后一个字符上

[3]译注： 严格说，它不是"e"行为的完全反向版，不管当前光标在当前word中的位置，它都会移动到前一个word的最后一个字符上

[4]译注： 汉字的空格不被视为空白字符

是把光标移动到当前行的第一个字符上. <Home>键也是如此. 图示如下:

```
                            Display
              ^
         <------------
    .....This is a line with example text
    <-----------------   --------------->
           0                  $
```

("....." 在这里代表空白)

"$"命令还可接受一个计数，就象其它的移动命令一样.但是移动到一行的行尾多于一次没有任何意义. 所以它的功能被赋予为移动到下一行的行尾. 如"1$"会将光标移动到当前行行尾，"2$"则会移动到下一行的行尾，如此类推.

"0"命令却不能接受类似这样的计数，因为"0"本身就是一个数字，所以合起来"0"会成为你前面指定的计数的一部分，另外，并不象其它命令一样可以举一反三，命令"^"前加上一个计数并没有任何效果[1].

03.3 移动到指定的字符上

一个最有用的移动命令是单字符搜索命令. 命令"fx"在当前行上查找下一个字符x. 提示: "f"意为"find".

例如，光标位于下行的开头，假如你要移动到单词human中的字符h上去. 只要执行命令"fh"就可以了:

```
                            Display
To err is human.  To really foul up you need a computer.
---------->---------------->
    fh           fy
```

上图同时展示了命令"fy"会将光标定位于单词really的尾部.

该命令可以带一个命令计数; 命令"3fl"会把光标定位于"foul" 的"l"上:

```
                            Display
To err is human.  To really foul up you need a computer.
          --------------------->
                  3fl
```

"F"命令向左方向搜索[2]:

[1]译注: 没有任何效果是说它与单个的"^"命令一样，并不是说光标根本不动
[2]译注: 规律:一个命令的大写字母形式会做同样的事情,但是方向相反

```
                              Display
To err is human.  To really foul up you need a computer.
          <----------------------
                    Fh
```

"tx"命令形同"fx"命令，只不过它不是把光标停留在被搜索字符上，而是在它之前的一个字符上. 提示: "t"意为"To". 该命令的反方向版是"Tx":

```
                              Display
To err is human.  To really foul up you need a computer.
           <------------  ------------->
                Th              tn
```

这4个命令都可以用";"来重复. 以","也是重复同样的命令，但是方向与原命令的方向相反[1]. 无论如何，这4个命令都不会使光标跑到其它行上去. 即使当前的句子还没有结束[2]

03.4 以匹配一个括号为目的的移动

写程序的时候很容易被层层嵌套的()给弄糊涂.此时使用命令"%[3]"就太方便了: 它跳转到与当前光标下的括号相匹配的那一个括号上去. 如果当前光标在"("上，它就向前跳转到与它匹配的")"上，如果当前在")"上，它就向后自动跳转到匹配的"("上去[4]:

```
                              Display
                              %
                            <----->
                    if (a == (b * c) / d)
                        <---------------->
                              %
```

这对方括号[]和花括号{}同样适用. (具体可以处理哪些括号可以由'matchpairs'选项来定义[5])

如果当前光标并没有停留在一个可用的括号字符上，"%"也会向前为它找到一个. 所以如果当前光标位于上例中的行首，"%"命令还是会向前先找到第一个"(",然后找到与它匹配的")":

---

[1]译注: 这意味着"Fx"本身是向左搜索，用","重复时因为反转了命令的方向，所以又变为向右搜索了

[2]译注: 对于中文用户来说，"fx"中的x也可以是一个汉字，按下"f"命令之后，打开中文输入法，输入一个汉字，注意只能是一个汉字，这要求你的Vim能将一个汉字当作一个字符来识别，在Windows平台上的预编译版本就可以做到这一点，其它几个以单字符为操作对象的同类命令也一样

[3]本文第一版中此处显示为 ",感谢钱震(qzhen@flotu.org)指出

[4]译注: 这种跳转当然可以跨行进行

[5]比如，还可以加入尖括号<>，这对HTML，XML的编写很有用

code
```
if (a == (b * c) / d)
---+---------------->
   %
```

03.5 移动到指定行

如果你是一个C或C++程序员，你应该很熟悉下面形式的编译器错误信息：

Display
```
prog.c:33: j   undeclared (first use in this function)
```

这行信息告诉你可能你要在第33行修改一些东西．那么你怎么找到第33行呢？一个办法是用命令"9999k"[1]然后用命令"32j"向下跳转32行． 这可不是个好办法，但是也能对付．一个更好的办法是用"G"命令[2]． 指定一个命令计数，这个命令就会把光标定位到由命令计数指定的行上． 比如"33G"就会把光标置于第33行上． （|usr_30.txt|中有更好的办法让你遍历编译器的错误信息列表，请参考:make命令的相关信息）

没有指定命令计数作为参数的话[3]，"G"会把光标定位到最后一行上．"gg"命令是跳转到第一行的快捷的方法． "1G"效果也是一样，但是敲起来就没那么顺手了．

Display
```
            |   first line of a file   ^
            |   text text text text    |
            |   text text text text    |  gg
        7G  |   text text text text    |
            |   text text text text
            |   text text text text
            V   text text text text    |
                text text text text    |  G
                text text text text    |
                last line of a file    V
```

[1]译注： 这个命令中的9999的意思是尽可能多地往上跳行，对于C/C++源程序来说，一般来说不会这么多行的源代码被放置在单个的源文件中，如果文件中当前行之前实际没有这么多行，Vim将会把光标置于第一行上． 这是作者谐趣的说法

[2]译注： 助记：G意为Go

[3]译注： 一般人的概念是参数出现在命令的后面，但Vim中的参数通指出现在命令之前或之后对命令起附加补充作用的所有信息，并且也不象你在使用命令行或写程序时的函数调用一样，这里的参数可能不以空格和逗号来分隔

另一个移动到某行的方法是在命令"%"之前指定一个命令计数[1]. 比如"50%"将会把光标定位在文件的中间[2]. "90%"跳到接近文件尾的地方[3].

上面的这些命令都假设你只是想跳转到文件中的某一行上，不管该行当前是否显示在屏幕上.但如果你只是想移动到目前显示在屏幕上的那些行呢? 下图展示了达到这一目标的几个命令:

Display
```
            +---------------------------+
H -->       | text sample text          |
            | sample text               |
            | text sample text          |
            | sample text               |
M -->       | text sample text          |
            | sample text               |
            | text sample text          |
            | sample text               |
L -->       | text sample text          |
            +---------------------------+
```

提示: "H"意为Home, "M"为Middle, "L"为Last.

03.6 告诉你当前的位置

要知道你当前在文件中的位置，共有三种方法:

1. 使用CTRL-G命令. 你会得到一些类似于下面的信息行(假设'ruler'选项已关闭):

Display
```
"usr_03.txt" line 233 of 650 --35%-- col 45-52
```

这行信息显示了你正在编辑的文件名，当前光标所在行的行号，总的行数，以及当前行所在文件中的百分比和当前光标所在的列的信息. 有时候你会看到两个以-分隔的数字来表示列，如"col 2-9". 这意味着你的光标位于第二个字符上，因为第一个字符是一个跳格键，占了8个字符的位置，所以屏幕上看起来该列位置是9.

2. 设置'number'选项[4]. 这会在每行的前面显示一个行号:

ex command
```
:set number
```

[1]译注:这里的命令前缀数字计数可不意味着对同一个命令重复执行N次
[2]译注: 意思很直观，文件的50%处
[3]译注: Vim对百分比的计算是以行为单位，不是字节数，如何跳转到以字节数为百分比或为偏移的字符上去? :goto 3
[4]译注: 译者建议大家总是打开该选项

号关闭，可以用命令[1]:

ex command
```
:set nonumber
```

因为'number'是一个二值选项，所以在它前面放一个"no"表示关闭该选项. 一个二值选项只有两种可能的值，开或关.

Vim有很多选项. 除二值选项外还有数字类型的选项和字符串类型的选项. 你会在接下来的例子中看到这些选项.

3. 设置'ruler'选项. 这会在Vim窗口的右下角显示当前光标位置

ex command
```
:set ruler
```

使用'ruler'选项有一个好处就是它不会占据太多的屏幕空间，你可以留出地方来给文本内容[2]

03.7 滚来滚去[3]

CTRL-U命令会使文本向下滚动半屏. 也可以想象为在显示文本的窗口向上滚动了半屏. 不要担心这种容易混淆的解释，不是只有你一个人搞不清楚.

CTRL-D命令将窗口向下移动半屏，所以相当于文本向上滚动了半屏:

[1]译注: 规律:no放置在boolean选项前面表示关闭该选项

[2]译注: 网络上有很多文档，以Vim查看时如果set number，则每行会超出屏幕少许，从而被折叠放到下一行上，看起来很不方便，这时就可以使用:set nonumber ruler，如果还是坚持想打开number，可以考虑重新格式化文本，请参考|gq|

[3]--<<腊笔小新>>

```
                          Display
                                   +-----------------+
                                   | some text       |
                                   | some text       |
                                   | some text       |
          +---------------+        | some text       |
          | some text     | CTRL-U  --> |            |
          |               |        | 123456          |
          | 123456        |        +-----------------+
          | 7890          |
          |               |        +-----------------+
          | example       | CTRL-D --> | 7890        |
          +---------------+        |                 |
                                   | example         |
                                   | example         |
                                   | example         |
                                   | example         |
                                   +-----------------+
```

要一次滚动一行可以使用CTRL-E(向上滚动)和CTRL-Y(向下滚动). 提示: CTRL-E 意为Extra. (如果你在用MS-Windows兼容的映射键, CTRL-Y可能被映射为重做而不是向下滚屏).

要向前滚动一整屏(实际上是整屏去两行)使用命令CTRL-F. 另外CTRL-B是它的反向版.很幸运CTRL-F是向前[1], CTRL-B是向后[2], 好记吧!

一个经常遇到的问题是你用"j"命令向下移动了若干行后当前光标已经处于屏幕的底端了. 而你又想查看当前行前后的几行内容. "zz"命令会把当前行置为屏幕正中央:

```
                          Display
          +------------------+         +------------------+
          | some text        |         | some text        |
          | some text        |         | some text        |
          | some text        |         | some text        |
          | some text        |  zz  --> | line with cursor |
          | some text        |         | some text        |
          | some text        |         | some text        |
          | line with cursor |         | some text        |
          +------------------+         +------------------+
```

"zt"命令会把当前行置于屏幕顶端[3], "zb"则把当前行置于屏幕底端[4].

[1]译注: 助记: Forward
[2]译注: 助记: Backward
[3]译注: "zt"中的t意为top, z字取其象形意义模拟一张纸的折叠及变形位置重置, 广泛用作折叠类命令的前辍, 特别注意zt不是Zhuan Tie(转贴)的缩写:-)
[4]译注: 助记: "b"意为bottom

此外还有一些与滚屏相关的命令，请参考|Q_sc|.若要一直保持当前行的前后都有一些内容显示在屏幕上，请参考'scrolloff'选项.

03.8 简单搜索

"/string"命令可用于搜索一个字符串. 比如要找到单词"include"，使用命令：

normal mode command

```
/include
```

你可能会注意到按下"/"键后光标跳到了[1]Vim窗口的最后一行，就象冒号命令行，你要查找的内容在这里键入. 在键入的过程中还可以用箭头键和删除键进行移动和修改.

最后按下回车键执行命令.

> 备注: 字符.*[]^%/?~$ 有特殊意义，如果你要找的东西包括这些内容，要在这些字符前面放置一个反斜杠. 见下文.

要查找上次查找的字符串的下一个位置. 使用"n"命令. 比如首先用下面命令找到光标之后的第一个#include:

normal mode command

```
/#include
```

接下来按几次"n". 你就会移动到接下来的几个#include 中去. 如果你知道你要找的确切位置是目标字符串的第几次出现，还可以在"n"之前放置一个命令计数. "3n"会去查找目标字符串的第3次出现. 在"/"命令前使用命令计数则不行[2].

"?"命令与"/"的工作相同，只是搜索方向相反[3]:

---

[1]译注: Vim中有几种情况光标会自动从一种模式跳到另一种模式，位置也因之改变

[2]译注: 这句话很容易引起误导，因为在normal模式下，先键入一个数字然后键入/继续输入要查找的字符，回车后可以向前找到第N个匹配处，其中N正是键入的数字，我估计这里说/之前的命令计数不生效是说，在第一次键入/text进行搜索之后，接下来按n或N只会找到下一个或前一个匹配，而不受/之前的N所影响，如:

```
a 1 b
a 2 b
a 3 b
a 4 b
a 5 b
a 6 b
```

如果normal模式下当前光标位于1上，则2/\d这个命令会直接找到数字3，中间的2被跳过去了，但按下来再按n查找下一个数字时，却不会再跳过4了

[3]译注: 规律: Vim命令一般都会对应有一个功能相同但方向相反的命令

normal mode command
```
?word
```

"N"命令会重复前一次查找，但是与最初用"/"或"?"指定的搜索方向相反．所以在"/"命令之后的"N"命令是向后搜索，而"?"之后的"N"命令是向前搜索．

忽略大小写

通常情况下你要准确地键入你要查找的东西．如果你并不关心目标字符中字母的大小写，可以通过设置'ignorecase'选项：

ex command
```
:set ignorecase
```

现在你再去搜索"word"，它就会同时匹配"Word"和"WORD"．　要回到对大小写的精确匹配,可以重设：

ex command
```
:set noignorecase
```

命令历史记录

假设你做过3次搜索：

normal mode command
```
/one
/two
/three
```

现在我们按下"/"来搜索，先别按回车键．如果此时你按下上箭头键，Vim会把"/three"放在命令行上，此时按下回车键就可查找"three"．　如果你不按回车键，而是继续按上箭头键，Vim就会把命令变为"/two"．　下一次是"/one"．

你同样可以用下箭头键来向下查找用过的搜索．

如果你知道你用过的某个搜索字串的开头，你就可以在键入这个开头部分之后再按上箭头键．比如上例中"/o<Up>"[1]Vim就会把"/one"放在命令行上．

以":"开始的命令也有一个历史记录．　它让你找到用过的冒号命令重复执行它．　这两个命令历史记录是相互独立的．

[1]译注：　<Up>代表你按下了上箭头键

在文本中查找下一个WORD

假设你在当前文件中有一个word是"TheLongFunctionName"，你想查找它的下一次出现在哪. 当然可以用"/TheLongFunctionName"，但这要敲太多次键盘. 万一哪个字符敲错了Vim就找不到你真正想要的东西.

有一个便捷的方法：把光标定位于这个word上，然后按下"*"键. Vim将会取当前光标所在的word并将它作用目标字符串进行搜索[1].

"#"命令是"*"的反向版. 还可以在这两个命令前加一个命令计数:"3*"查找当前光标下的word的第三次出现.

查找整个WORD

如果你用"/the"来查找Vim也会匹配到"there". 要查找作为独立单词的"the"使用如下命令:

normal mode command

```
/the\>
```

"\"是一个特殊的记法，它只匹配一个word的结束处.近似地，"\"匹配到一个word的开始处[2]. 这样查找作为一个word的"the"就可以用:

normal mode command

```
/\<the\>
```

这个命令就不会匹配到"there"或"soothe".注意"*"和"#"命令会在内部使用这些标记word开始和结束的特殊标记来查找整个的word(你可以用"g*"和"g#"命令来同时匹配那些包含在其它word中的字串.)

高亮显示搜索结果

如果你在编辑一段源程序时看到了一个叫"nr"的变量. 你想查看一下这个变量就被用在了哪些地方. 简单的办法是把光标移到"nr"上然后用"*" 命令和"n"命令一个一个地查找所有的匹配.

不过还有更好的办法. 使用下面的命令:

ex command

```
:set hlsearch
```

[1]译注: 问题:但如果要匹配一小片包含了几个word的文本呢?如何避免手工键入?答案: Visual select, yank, :let @/=@", n

[2]译注: 一个word的结束处和开始处仅指一个位置，本身不占据任何字符宽度,或者说，它占据的字符宽度是0

现在你要再查找"nr"，Vim就会以某种形式高亮显示所有符合的匹配． 对于想查看一个变量被用在哪些地方，这个办法太棒了，不需任何其它的命令[1]

看得眼花的时候还可以关闭这一功能：

ex command
```
:set nohlsearch
```

不过你要在下次搜索时再次使用这一功能就又得打开它． 如果你只是想去掉当前的高亮显示，可以使用下面的命令[2]：

ex command
```
:nohlsearch
```

这不会重置'hlsearch'选项的值．它只是关闭了该语法项高亮显示． 一旦你再次执行一个搜索命令,被匹配到的目标就又会以高亮形式显示了．"n"和"N"命令也一样会引起高亮显示．

调理搜索命令

有一些选项用来改变搜索命令的工作方式． 下面是一些最常用的：

ex command
```
:set incsearch
```

这使得你在键入目标字符串的过程中Vim就同时开始了搜索工作．使用这种方法可以让你在尚未完全键入字串时就能找到目标．你可以选择按回车跳转到当前匹配到的位置或者键入字串的其它部分继续搜索：

ex command
```
:set nowrapscan
```

该设置会使搜索过程在文件结束时就停止．或者，在你反向搜索时在到达文件开头时停止． 'wrapscan'选项的默认值是开，这样搜索在达到文件的头尾时都会绕向另一个方向继续进行．

INTERMEZZO
[3]

如果你感到前面讲到的这些选项很好使，并且每次用Vim时都去设置它们的值，你完全可以把这些设置命令放在Vim的启动文件里． 修改该文件时请遵照|not-compatible|里的建议．通过下面的命令可以找到它的位置：

---

[1]译A：如果排除函数外别处同名变量的干扰呢？ 答：|/\%<31| 限定

[2]译注： 这里说的去掉当前的高亮只是使被匹配的文本不再以区别于普通文本的形式显示，既不是删除相应的文本，也不是改变'hlsearch'选项的值

[3]译注： 这个破词实在没法翻译了

ex command
```
:scriptnames
```

例如，象这样去编辑该文件:

ex command
```
:edit ~/.vimrc
```

接下来你就可以在里面添加你自己的命令设置了，象就你在Vim里进行设置时所用的命令一样. 如:

ex command
```
Go:set hlsearch<Esc>
```

"G"命令先移动到文件末尾. "o"另辟一行进行编辑，在该行上键入你的":set"命令. 然后用<Esc>来退出插入模式. 最后保存文件:

normal mode command
```
ZZ
```

下次启动Vim时，'hlsearch'选项就是打开的了.

03.9 简单的搜索模式

Vim用正则表达式来描述要查找的目标. 正则表达式在描述一个搜索模式方面功能超强，语法紧凑. 但是，要运用这种强大的功能是要付出代价的，因为正则表达式用起来还是需要一些技巧的.

本节中我们将只涉及一些最基本的内容. 关于该主题更多的内容请查看|usr_27.txt|.你也可以在|pattern|找到它的完整描述.

一行的开头与结尾

^ 字符匹配一行的开头. 在标准的美语键盘上你会发现它在数字键[1]6的上面. 象"include"这样的模式可以匹配出现在一行中任何位置的include这个单词.但是模式"^include"就只匹配出现在一行开头的include.

$字符匹配一行的末尾. 所以"was$"只匹配位于一行末尾的单词was.

下面我们用字符"x"来标记模式"the"匹配到下行中的哪些地方:

Display
```
the solder holding one of the chips melted and the
xxx                          xxx                 xxx
```

[1]译注: 不是小键盘哦

而使用"/the$"时匹配的结果则是:

Display
```
the solder holding one of the chips melted and the
                                               xxx
```

用"^the"找到的则是:

Display
```
the solder holding one of the chips melted and the
xxx
```

你也可以试一下"/^the$"会怎么样，它只会匹配到一行的内容仅包含"the"的情况. 有空白字符也不行，所以如果有一行的内容是"the "，那么这个匹配同样不会成功.

匹配任何的单字符

.这个字符可以匹配到任何字符. 比如"c.m"可以匹配任何前一个字符是c后一个字符是m的情况，不管中间的字符是什么[1]. 如:

Display
```
We use a computer that became the cummin winter.
         xxx              xxx      xxx
```

匹配特殊字符

如果你要查找的东西本身就是一个.号呢，这时你就要想办法去掉.号在正则表达式里的特殊意义了: 放一个反斜杠在它前面. 如果你用"ter."来查找，你会找到如下的匹配:

Display
```
We use a computer that became the cummin winter.
              xxxx                          xxxx
```

而使用"ter\."你就只会找到上面第2个匹配了.

03.10 使用标记

当你用"G"命令从一个地方跳转到另一个地方时，Vim会记得你起跳的位置. 这个位置在Vim中是一个标记. 使用下面的命令可以使你跳回到你刚才的出发点:

[1]译注: 对于中文用户，如果你用l命令移动光标时单位是一个汉字，那么.可以匹配一整个汉字，否则它只会匹配半个汉字，另外，严格说，.匹配除换行符外的任何字符

normal mode command
```
``
```

这个符号看起来象是一个反方向的单引号，或者，叫它开单引号[1]

再次使用上面的这个命令你就会又跳回来了． 这是因为`也是一个跳转命令，所以上次跳转时的起跳位置现在被标记为了`．

更一般地说，只要你执行一个命令使光标定位于当前行之外的某行上去，这都叫一个跳转． 包括"/"和"n"这些搜索命令(不管被找到的东西离当前位置有多远)． 但是字符搜索命令"fx"和"tx"，或者是以word为单位的移动光标位置的命令"w"和"e"不叫跳转．

同时，"j"和"k"命令并不被视为一个跳转，即使你在它们之前加了命令计数让当前光标跳到老远的地方也是如此．

``命令可以在两点之间来回跳转． CTRL-O命令是跳转到你更早些时间停置光标的位置(提示： O意为older)． CTRL-I则是跳回到后来停置光标的更新的位置(提示： I在键盘上位于O前面)． 考虑一下以下面顺序执行这3个命令会怎样：

normal mode command
```
33G
/^The
CTRL-O
```

首先你会跳转到33行[2]，然后向下搜索以"The"开头的目标．接下来的CTRL-O会让你跳回到33行． 再一个CTRL-O又让你跳回到执行"33G"命令之前的位置．现在再用CTRL-I命令的话你又会再次回到第33行．再一个CTRL-I又会让你回到刚才找到的匹配"/^The"的那一行：

Display
```
         |  example text   ^             |
  33G    |  example text   |  CTRL-O     | CTRL-I
         |  example text   |             |
         V  line 33 text   ^             V
         |  example text   |             |
 /^The   |  example text   |  CTRL-O     | CTRL-I
         V  There you are  |             V
            example text
```

备注: 使用CTRL-I 与按下<Tab>键一样．

[1]译注： 相对地，'就是一个闭单引号，提示： 在标准键盘布局上，字符`位于数字1的左边

[2]译：在讲述与光标有关的主题时，有时用"你"跳转到某某处，当然都是指光标的跳转

":jumps"命令会列出关于你曾经跳转过的位置的列表. 你最后一个跳转的位置被特别以一个">"号标记.

有名标记

Vim允许你在文本中定义你自己的标记. 命令"ma"将当前光标下的位置名之为标记"a".从a到z一共可以使用26个自定义的标记. 定义后的标记在屏幕上也看不出来. 不过Vim在内部记录了它们所代表的位置.

要跳转到一个你定义过的标记, 使用命令`{mark}, {mark}就是你定义的标记的名字.就象这样:

normal mode command

```
`a
```

命令'mark(单引号, 或者叫呼应单引号)会使你跳转到mark所在行的行首.这与`mark略有不同, `mark会精准地把你带到你定义mark时所在的行和列.

标记对于编辑那些有两块内容相互关联的文件十分有用. 想象一下你在文件开头有一段文字需要时时参考, 但实际上要修改编辑的地方却在文件结尾处的情形.

你可以移动到文件开始处并在此放置一个名为s(start)的标记:

normal mode command

```
ms
```

然后你可以转移到你需要编辑的地方并在此命名一个叫e(end)的标记:

normal mode command

```
me
```

现在你就可以在两地之间自由移动了, 若要参考文件开头的部分:

normal mode command

```
's
```

然后你可以用''命令跳转回刚才正在编辑的地方, 或者用'e跳转到定义标记e的文件结尾处.

这里用s代表文件开头[1], 用e代表文件结尾[2]可并不说它们有任何特别之处, 只是为了方便记忆而已.

你也可以使用下面这个命令来查看关于标记的列表:

[1]译注: start
[2]译注: end

ex command
```
:marks
```

在这个列表里你会看到一些特别的标记. 象下面这些:

List

'	进行此次跳转之前的起跳点
"	上次编辑该文件时光标最后停留的位置
[	最后一次修改的起始位置
]	最后一次修改的结束位置

下一章: |usr_04.txt| 小幅改动
版　权: 请参考|manual-copyright| vim:tw=78:ts=8:ft=help:norl:

usr_04.txt Vim 6.3版.最后修改: 2004年01月17日

VIM用户手册--- 作者: Bram Moolenaar

小幅改动

天下大事，必作于细。

---老子≪道德经≫

本章向你展示几种移动文本和作出更改的方法. 你会学到3种改变文本的基本方法: 操作符命令和位移, Visual模式和文本对象.

下一章: |usr_05.txt| 定制你的Vim
前一章: |usr_03.txt| 移动
目 录: |usr_toc.txt|

04.1 操作符命令和位移

在第2章中你已经知道"x"命令可以删除一个字符. 使用一个命令记数"4x"可以删除4个字符.

"dw"命令可以删除一个word. 你可以把其中的"w"看作是向右移一个word的命令.事实上, "d"命令可以后跟任何一个位移命令, 它将删除从当前光标起到位移的终点处的文本内容.

举例来说, "4w"命令是向前移动4个word. 所以"d4w"命令是删除4个word.

```
                         Display
        To err is human. To really foul up you need a computer.
                         ------------------>
                                 d4w

        To err is human. you need a computer.
```

Vim只删除到位移命令之后光标的前一个位置. 这是因为Vim知道你并不是要删除下一个word的第一个字符. 如果你用"e"命令来移动到word的末尾, Vim也会假设你是要包括那最后一个字符[1]:

```
                         Display
        To err is human. you need a computer.
                        -------->
                           d2e

        To err is human. a computer.
```

删除的内容是否包括光标所移动到的那个字符上取决于你的位移命令. 在联机参考手册上把这种不包括该位置的操作叫做"排外的", 把包括该位置的操作叫"内包的".

"$"命令是移动光标到行尾. 所以"d$"命令就是删除自当前光标到行尾的内容. 这是一个"内含的"位移, 所以该行最后一个字符也被删除:

```
                         Display
        To err is human. a computer.
                       ------------>
                            d$

        To err is human
```

此类命令有一个固定的模式: 操作符命令+位移命令. 首先键入一个操作符命令. 比如"d"是一个删除操作符. 接下来是一个位移命令如"4l"或"w". 这样你可以以任何移动光标命令所及之处, 都是命令的作用范围.

04.2 改变文本

另一个操作符命令是"c", 改变命令. 它的行为与"d"命令类似, 不过在命令执行后会进入Insert模式. 比如"cw"改变一个word. 或者, 更准确地说, 它删除一个word并让你置身于Insert模式:

[1]译注: 所幸, 这种假设对绝大多数人来说是正确的

Display

```
To err is human
   -------->
      c2wbe<Esc>

To be human
```

这里的"c2wbe<Esc>"包含了下面的命令元素:

List

c	修改操作符
2w	移动两个word(它们将被删除并从此开始Insert模式)
be	插入这两个字符
<Esc>	回到Normal模式

如果你留心的话也许已经注意到这里面有一些奇怪的事情:"human"之前的空格并没有被删除.就象那句谚语里说的: 对每一个问题,都会有一个简单而清晰的答案,而那个答案总是错的. 这正是"cw"命令的情况."c"操作符与d操作符一样,只是有一个例外:"cw",它就象"ce"一样,改变到一直到word结尾的内容.而word之后的空格被留下了. 这个例外可以一直追溯到古老的Vi编辑器. 因为多数人已经习惯了,所以Vim里这个例外也被保留下来了[1].

更多的更改

就象"dd"命令可以删除整行一样,"cc"命令可以改变整行. 不过仍保持原来的缩进(一行打头的空白).

也正如"d$"删除到行尾为止的内容,"c$"改变当前光标到行尾的内容.就好象是用"d$"删除然后又以"a"进入Insert模式追加新的文本一样.

快捷命令

有一些操作符+位移命令使用率是如此之高以至于它们以一个单独的字符作为其快捷方式:

List

x	代表dl(删除当前光标下的字符)
X	代表dh(删除当前光标左边的字符)
D	代表d$(删除到行尾的内容)
C	代表c$(修改到行尾的内容)
s	代表cl(修改一个字符)
S	代表cc(修改一整行)

[1]译者: 规律:每条规律都有例外

命令记数放在哪

命令"3dw"和"d3w"都是删除3个word. 如果你真要钻牛角尖的话，第一个命令"3dw"可以看作是删除一个word的操作执行3次；第二个命令"d3w"是一次删除3个word. 这是其中不明显的差异. 事实上你可以在两处都放上命令记数，比如，"3d2w"是删除两个word，重复执行3次，总共是6个word.

替换单个字符

"r"命令不是一个操作符命令. 它等待你键入下一个字符用以替换当前光标下的那个字符.你也可以用"cl"或"s"完成同样的事情，但用"r"的话就不需要再用<Esc>键回到Normal模式了.

Display
```
there is somerhing grong here
rT           rt    rw

There is something wrong here
```

"r"命令前缀以一个命令记数是将多个字符都替换为即将输入的那个字符.

Display
```
There is something wrong here
                   5rx

There is something xxxxx here
```

要把一个字符替换为一个换行符使用"r<Enter>". 它会删除一个字符并插入一个换行符. 在此处使用命令记数只会删除指定个数的字符: "4r<Enter>"将把4个字符替换为一个换行符[1].

04.3 重复改动

"."命令是Vim中一个简单而强大的命令. 它会重复上一次做出的改动. 例如，假设你在编辑一个HTML文件，想删除其中所有的<B>标签. 你把光标置于<B>的<字符上然后命令"df>". 然后到</B>的<上用"."命令做同样的事. "."命令会执行上一次所执行的更改命令(此例中是"df>"). 要删除另一个标签，同样把光标置于<字符上然后执行"."命令即可.

[1]译：规律：通常的规律延申至无实际意义时，将打破规律

Display
```
                                        To <B>generate</B> a table of <B>contents
f<    找到第一个 <      --->
df>   删除到 >处的内容     -->
f<    找到下一个 <          --------->
.     重复 df>                       --->
f<    找到下一个 <                      ------------->
.     重复 df>                                      -->
```

"."命令会重复你做出的所有修改，除了"u"命令CTRL-R和以冒号开头的命令．（译者： "."需要在Normal模式下执行，它重复的是命令，而不是被改动的内容，如下两行文本：

Display
```
asdf 123
asdf 1234
```

光标置于第一行的1上时执行了"cwxyz"，然后退回到Normal模式，此时第一行变为：

Display
```
asdf xyz
```

标置于第二行的1上，执行.命令，则第二行将变为：

Display
```
asdf xyz
```

而不是：

Display
```
asdf xyz4
```

因为真正重复的是命令，而不是从字面上看到的将3个字符换为xyz.）

另一个例子： 你想把"four"改为"five"． 它在你的文件里多次出现． 你可以用以下命令来做出修改：

List
```
/four<Enter>     找到第一个字符串"four"
cwfive<Esc>      把它改为"five"
n                找到下一个字符串"four"
.                同样改为"five"
n                继续找下一个
.                做同样的修改
                 等等
```

04.4 Visual模式

删除那些简单的文本对象用操作符命令+位移命令就足够了. 但是通常很难说用什么位移命令可以把光标刚好移动到你想删除的文本范围. 这时你可以用Visual模式.

按"v"可以进入Visual模式. 移动光标以覆盖你想操纵的文本范围. 同时被选中的文本会以高亮显示. 最后键入操作符命令.

例如, 要删除一个单词的后半部分至下一个单词的前半部分:

Display

```
This is an examination sample of visual mode
               ---------->
                 velllld

This is an example of visual mode
```

要做这样的修改你不必去计算要按多少次"l"才能刚好达到想要的位置. 在你按下"d"命令时就可准确看到哪些文本将会被删除.

发出实际的更改命令之前任何时间你都可以决定放弃, 用<Esc>命令退出Visual模式. 一切都象没发生过一样.

选择多行

如果你想整行整行地操纵文本, 使用"V"进入Visual模式. 你会看到被选中的文本是是一整行一整行为单位. 命令左右移动命令毫无意义. 而上下位移命令则会整行整行地选择文本.

如下例中, 用"Vjj"命令选中3行:

Display

```
                          +------------------------+
                          | text more text         |
                       >> | more text more text    | |
        selected lines >> | text text text         | | Vjj
                       >> | text more              | V
                          | more text more         |
                          +------------------------+
```

选择文本块

如果你想以一个矩形的文本块为对象进行操作, 你需要用CTRL-V进入Visual模式. 在编辑表格时这可就派上用场了.

Display

```
name      Q1    Q2    Q3
pierre    123   455   234
john      0     90    39
steve     392   63    334
```

要删除其中的"Q2"列，把光标置于"Q2"的"Q"上. 按下CTRL-V进入文本块Visual模式.现在可以用"3j"向下移动3 行，用"w"选择直到下一个word的区域. 你可以看到被选中的文本中包含了下一列的第一个字符. 使用"h"排除这一列. 现在按下"d"中间的这一列就被删除了.

到另一端

如果你已经在Visual模式下选中了一些文本，但此时发现还要改变另一头的被选择区域，"o"命令(提示: o 代表other end另一头)会让光标置于被选中文本的另一头，这样你就可以通过控制光标移动来决定被选文本将从何处开始. 再按"o"又会让光标置于被选文本的末端.

当你进行矩形文本块内容的选择时，你有4个角都可以改变. "o"只会把你带到对角的位置去，使用"O"命令可以让你在同一行的左右两个角之间移动[1].

注意"o"和"O"在Visual模式与Normal模式下行为迥异，在Normal模式下它们是在当前行的下面或上面插入一个新行.

04.5 移动文本

你以"d"或"x"这样的命令删除文本时，被删除的内容还是被保存了起来. 你还可以用p命令把它取回来(在Vim中这叫put)

看一下这是如何工作的. 首先你删除一整行内容，把光标置于该行键入"dd". 现在移动光标到想放入该的地方键入"p"命令. 这样该行就被插入到当前光标下面了.

Display

```
a line          a line          a line
line 2    dd    line 3     p    line 3
line 3                          line 2
```

因为你删除的是整行的内容，所以"p"命令把整个文本行都放到光标下面作为单独一行. 如果你删除的是一行的部分内容(比如说一个word)，"p"命令就会把这部分文本放到当前光标后面[2].

[1]译者: 你应该知道如何在4个角之间移动
[2]译: 不会因此多出一个新行

```
                              Display
         Some more boring try text to out commands.
                          ---->
                           dw

         Some more boring text to out commands.
                          --------->
                             welp

         Some more boring text to try out commands.
```

关于PUTTING的更多内容

"P"命令与"p"一样取回被删除的内容，不过它把被取回的内容置于光标之前． 对于以"dd"删除的整行内容，"P"会把它置于当前行的上一行．对于以"dw"删除的部分内容，"P"会把它放回到光标之前[1]．

你可以多次取回被删除的内容．其内容取之不竭．

也可以对命令"p"和"P"命令使用命令记数． 它的效果是同样的内容被取回指定的次数.这样一来"dd"之后的"3p"就可以把被删除行的3 份副本放到当前位置．

交换两个字符

输入文本的时候，人们常常会发现手比脑跑得要快(或者脑比手跑得快)． 不管谁更快结果都是拼错字，比如把"the"拼成"teh"． 在Vim中改正此类错误十分容易，把光标置于"teh"的e上执行命令"xp"． 它的工作如下： "x"删除字符e并把它放入一个寄存器中． "p"命令把被删除的e再放回到当前光标之后，也就是h后面．

```
                              Display
         teh       th       the
          x         p
```

04.6 复制文本

要把文本内容从一处复制到另一处，你可以先删除它，然后马上用"u"命令恢复删除.再用"p"把它放到你想要的地方去． 不过做这件事还有另一种更快的方法： yanking [2]． "y"操作符命令会把文本复制到一个寄存器中． 然后可以用"p"命令把它取回[3]．

[1]译：即光标左边
[2]译者： term:yanking
[3]译者： 所谓"一个寄存器"是指默认的寄存器"

Yanking只是Vim对复制的另一种说法，"c"字母已经用来表示更改操作符了[1]，"y"还没人占用． 把这个操作符叫做"yank" 也会时时提醒你记得用"y"操作符．

因为"y"是一个操作符命令，所以你可以用"yw"来复制一个word． 同样可以使用命令记数．如下例中用"y2w"命令复制两个word:

```
Display
        let sqr = LongVariable *
                 -------------->
                       y2w

        let sqr = LongVariable *
                               p

        let sqr = LongVariable * LongVariable
```

注意"yw"复制的内容中包括了word之后的空白字符．如果你不想要它们，那就用"ye"．

"yy"命令复制一整行，就象"dd"是删除一整行一样． 不过并不象"D"删除当前光标至行尾的内容那样，"Y"也是复制整行的内容．注意这种规律中的例外！ 复制当前光标至行尾的命令是"y$"．

```
Display
a text line    yy        a text line               a text line
line 2                   line 2             p      line 2
last line                last line                 a text line
                                                   last line
```

04.7 使用剪贴板

如果你用的是Vim的GUI版本(gvim)，你会在"Edit"菜单中发现"Copy"命令． 首先在Visual模式下选择一些文本，然后用Edit/Copy菜单． 现在被选择的文本就被复制到了剪贴板.这样你就可以在其它程序里粘贴这些内容了． 当然也可以在Vim里面使用[2]．

如果你在其它应用程序里把文本内容复制到了剪贴板，也可以用Vim的Edit/Paste菜单把它粘贴过来． 这在Normal模式和Insert模式下都可以． 在Visual模式下被选中的文本就会被粘贴进来的内容给替换掉．

[1]译者:代表单词change，所以不能再代表copy了

[2]译：从技术上说，此处的Copy命令与Normal模式下的yank命令区别在于工具栏或菜单中的Copy是把内容复制到了各应用程序共享的公用剪贴板上，Vim 中对应的寄存器是*,而y命令则把文本对象复制到了Vim内部的默认寄存器上"上，它是Vim私有的

"Cut"菜单命令会在把文本放到剪贴板之前先将其删除. "Copy", "Cut"和"Paste"菜单命令可从上下文菜单中选取(当然前提是要有上下文菜单才行). 如果你的Vim有一个工具栏的话, 你应该也能在那里找到它们对应的小图标.

如果你没用GUI, 或者你不喜欢用菜单, 你也可以用另一种方法来做同样的事. 使用通常的"y"(yank)和"p"(put)命令, 不过在命令之前附加一个"*(一个双引号, 紧挨着是一个星). 要把一行内容复制到剪贴板:

normal mode command
```
"*yy
```

要把剪贴板的内容复制过来:

normal mode command
```
"*p
```

这些功能只有Vim支持剪贴板操作时才可用. 关于剪贴板操作的更多内容请参考|09.3|和|clipboard|.

04.8 文本对象

如果光标位于一个单词的中间而你要删除这个单词, 通常你需要把光标移到该单词的开头然后用"dw"命令. 不过有一个更简单的办法: ".daw" [1]

Display
```
this is some example text.
              daw

this is some text.
```

"daw"中的"d"还是操作符命令. "aw"是一个文本对象. 提示: "aw"意为"A Word". 这样"daw"的完整意思是"Delete A Word", 更准确地说,该Word之后的空白字符也被删除了(即位于行末尾的单词之前的空白).

使用文本对象是在Vim中更改文本的第三种方法. 我们已经介绍过操作符+位移命令和Visual模式了. 现在来看一下操作符命令+文本对象.

它很象操作符+位移命令, 但是它的起始点不象前者一样始于当前光标, 终于位移命令. 它不管当前光标所在的位置而把整个文本对象作为操作对象.

要修改一整个句子使用命令"cis". 以下面的文本为例:

[1]译者: Vim也赞成PERL的哲学: There is more than one way to do the same thing

Display
```
        Hello there.  This
        is an example.  Just
        some text.
```

现在把光标移到第二行的"is an"上. 使用"cis"命令:

Display
```
        Hello there.    Just
        some text.
```

光标被置于第一行中的空白位置. 现在你可以在此键入新的句子"Another line.":

Display
```
        Hello there.  Another line.  Just
        some text.
```

"cis"由操作符"c"和文本对象"is"组成. 它是"Inner Sentence"的缩写. 相应地还有一个叫"as"(a sentence)的文本对象. 两者的不同是"as"包含了句子后面的空格而"is"不包括.如果你想删除一个句子, 你会希望把它后面的空白也给删除, 所以此时最好用"das".如果你是想以新的文本替代它, 空白就可以留下来, 使用"cis"好了.

你也可以在Visual模式使用文本对象. 它将把指定的文本对象选进Visual模式的文本选择区域中. 当前模式仍是Visual模式, 所以你可以多次使用该命令. 例如, 以"v"开始Visual模式, 以"as"选取一个句子. 现在你可以重复使用"as"来包括进更多的句子. 最后你可以用一个操作符命令来作用于最终被选中的范围.

你可以在|text-objects|处发现一个很长的文本对象列表.

04.9 替换模式

"R"命令会让Vim进入replace模式. 在此模式下, 每个键入的字符都会替换掉当前光标下的字符. 直到你键入<Esc>结束该模式[1].

下例中你可以在"text"的"t"上开始进入Replace模式:

Display
```
        This is text.
                Rinteresting.<Esc>

        This is interesting.
```

[1]译者: replace模式下的例外是按下回车键并不会把当前字符替换为回车, 而是插入一个回车

也许你已经注意到该命令实际替换掉了原来的5个字符．后面还有12个字符．"R"命令会在无字符可替换时继续拓展该行以容纳更多你要键入的内容．不过并不会延续到下一行[1]．

你可以用<Insert>键来在Insert模式和Replace模式之间来回切换．[2]你按下<BS>键作出修改时，你会发现原来的字符又回来了． 所以它对于最后键入的字符来说实际上等价于一个撤消操作．

04.10 结论

操作符命令，位移命令和文本对象可以让你在使用这些命令时任意组合．现在你已经知道它们是如何工作的了，你可以使用操作符命令N配上位移命令M来构造一个N*M命令了！

你可以在|operator|找到一个操作符命令的列表．

比如，有多种办法可以删除文本． 下面是一些常用的方法：

List

x	删除当前光标下的字符("dl"的快捷命令)
X	删除当前光标之前的字符("dh"的快捷命令)
D	删除自当前光标至行尾的内容("d$"的快捷命令)
dw	删除自当前光标至下一个word的开头
db	删除自当前光标至前一个word的开始
diw	删除当前光标所在的word(不包括空白字符)
daw	删除当前光标所在的word(包括空白字符)
dG	删除当前行至文件尾的内容
dgg	删除当前行至文件头的内容

如果你用"c"命令代替"d"这些命令就都变成更改命令． 使用"y"就是yank命令，如此类推．

此外还有一些不太常用的用于更改文本内容的命令：

List

~	改变当前光标下字符的大小写，并将光标移至下一个字符．这不是一个操作符命令(除非你设置了'tildeop' [3]选项)，所以你不能让它与一个位移命令搭配使用．但它可以在Visual模式下改变所有被选中的文本的大小写．
I	将光标置于当前行第一个非空白字符处并进入Insert模式
A	当光标置于当前行尾并进入Insert模式

---

[1]译者： 12是怎么计算出来的??
[2]译注： 在Replace模式下

下一章：|usr_05.txt| 定制你的Vim
版　权：　请参考|manual-copyright| vim:tw=78:ts=8:ft=help:norl:

usr_05.txt Vim 6.3版.最后修改：2004年03月12日

VIM用户手册--- 作者： Bram Moolenaar

定制你的Vim

Vim可以根据每个人的喜好进行调整． 本章向你展示如何对Vim进行不同的设置．如何增加plugin来扩展它的功能．以及如何定义你自己的宏．

下一章： |usr_06.txt| 使用语法高亮
前一章： |usr_04.txt| 进行小幅改动
目 录： |usr_toc.txt|

05.1 vimrc文件 *vimrc-intro*

也许你早已厌倦于手工键入那些常用的命令． 要使你喜好的选项和映射一次性准备就绪，你可以把它们统统写进一个叫vimrc的文件．Vim在启动时会读取该文件．

如果你找不到你的vimrc的具体名字和位置，使用这个命令：

ex command
```
:scriptnames
```

命令输出的第一行会有一个叫".vimrc"或"_vimrc"的文件，位置是你的home目录．

如果你现在还没有自己的vimrc文件，请参考|vimrc|中的建议，看看你可以在哪里创建自己的vimrc文件．同时":version"命令也会列出Vim是在哪些目录寻找该文件的．

对Unix系统而言这个文件一般是：

List
```
~/.vimrc
```

对MS-DOS和MS-Windows来说通常是:

List
```
$HOME/_vimrc
$VIM/_vimrc
```

vimrc文件里可以包含任何你可以在冒行命令行上使用的命令. 最简单的命令是对选项的设置. 比如你想在使用Vim时总是打开'incsearch'选项. 就可以把下面这一行加进你的vimrc文件:

ex command
```
set incsearch
```

要使它自动生效你还得退出Vim等到它下一次启动. 本文稍后会告诉你如何在不退出Vim的情况下使用之生效.

本章只讲述基本的设置问题, 要全面了解如何写一个vim脚本请参考|usr_41.txt|.

05.2 vimrc示例 *vimrc_example.vim*

第一章里我们提到如何使用vimrc的示例文件(该文件随Vim一起发布)让Vim工作于not-compatible模式(请参考|not-compatible|). 这个示例文件的位置是:

List
```
$VIMRUNTIME/vimrc_example.vim
```

本节中我们会讲解在该文件中使用的一些命令. 你可以从中学习如何设置自己偏好的选项.虽然这里不会逐条讲解里面的每条命令. 但是你可以用":help"命令来学习更多的东西.

ex command
```
set nocompatible
```

就象在第一章中提到的, 本手册中对Vim的描述都假设它是在增强模式下, 所以并不完全与Vi兼容. 所以首先确保关闭了'compatible'选项.

ex command
```
set backspace=indent,eol,start
```

这条命令告诉Vim在Insert模式下退格键何时可以删除光标之前的字符．选项中以逗号分隔的三项内容分别指定了Vim可以删除位于行首的空格，断行，以及开始进入Insert模式之前的位置．

ex command
```
set autoindent
```

这个命令让Vim在开始一个新行时对该行施以上一行的缩进方式．这样，你在Insert模式下按回车或在Normal模式下按o来添加新行时该行将会与上一行有相同的缩进．

ex command
```
if has("vms")
  set nobackup
else
  set backup
endif
```

这段脚本告诉Vim在覆盖一个文件之前备份该文件．但是对VMS系统除外，因为该系统已经为文件保存了老的版本．备份文件名由当前文件名加后辍"~"组成．请参考|07.4|．

ex command
```
set history=50
```

设置冒号命令和搜索命令的命令历史列表的长度．当然你也可以设置其它的值．

ex command
```
set ruler
```

总是在Vim窗口的右下角显示当前光标的行列信息．

ex command
```
set showcmd
```

在Vim窗口的右下角显示一个完整的命令已经完成的部分．比如说你键入"2f"，Vim就会在你键入下一个要查找的字符之前显示已经键入的"2f"．一旦你接下来再键入一个字符比如"w"，那么一个完整的命令"2fw"就会被Vim 执行，同时刚才显示的"2f"也将消失．

Display
```
+--------------------------------------------------+
|text in the Vim window                            |
|~                                                 |
|~                                                 |
|-- VISUAL --                   2f     43,8   17%  |
+--------------------------------------------------+
 ^^^^^^^^^^^                  ^^^^^^^^ ^^^^^^^^^^
  'showmode'                  'showcmd'  'ruler'

set incsearch
```

在你键入要搜索的字串的同时就开始搜索当前已经键入的部分[1]..

ex command
```
map Q gq
```

该命令定义了一个键映射. 这一主题的更多内容在下一节. 这里的这个命令定义了一个"Q"命令映射到"gq"操作符命令. 其实Vim5.0版以前"Q" 本身即是这样的一个命令. 现在如果没有这一映射, "Q"命令会进入Ex模式, 一般情况下你不需要进入这种模式.

ex command
```
vnoremap p <Esc>:let current_reg = @"<CR>gvs<C-R>=current_reg<CR><Esc>
```

这是一个复杂的映射. 这里不涉入对更多细节的描述. 它的功能是让"p"可以在Visual模式下以此前yank的内容替换当前选择的文本块. 从这个例子你可以了解到映射也可以用来实现一些复杂的操作. 但是本质上, 它所做的与你手工连续键入这些命令毫无二致.

ex command
```
if &t_Co > 2 || has("gui_running")
  syntax on
  set hlsearch
endif
```

这段脚本打开语法高亮功能, 前提是当前系统支持彩色显示. 'hlsearch'告诉Vim高亮显示所有与最后一次搜索目标串相匹配的文本. "if"命令经常用于这种满足某个条件才设置一个选项的情境. 关于如何写Vim脚本的更多内容请参考|usr_41.txt|.

[1]译:对一个技巧的灵活运用与将之准确地以文字描述之间是个不可逆的过程. 一个人总可以以它自己熟悉的形式将自己头脑中事物十分清楚地描述出来, 但这种描述对于一个完全不了解该事物的人来说, 几乎完全是无济于事, 浪费的时间也白搭. 所以最好还是你亲自去试一试, 尤其是尝试的机会与成本都允许的情况下, 对于电脑知识的学习, 大多数情况下, 你有一台电脑, 有了软件环境就万事俱备了

vimrc-filetype

ex command
```
filetype plugin indent on
```

这个命令开启了Vim的三种智能:

1. 自动识别文件类型

   你开始编辑一个文件时，Vim就会自动识别它是何种类型的文件. 比如说你打开了"main.c"，Vim就会根据它的".c"扩展名知道它是一个类型为"c"的C语言源程序文件. 当你编辑一个文件其第一行是"#!/bin/sh"时，Vim又可以据此判断它是一个类型为"sh"的shell脚本文件.

2. 用文件类型plugin脚本

   不同的文件类型需要搭配适合于它[1]的编辑选项. 比如说你在编辑一个"c"文件，那么打开'cindent'就非常有用. 这些对某种文件类型来说最常用的选项可以放在一个Vim中叫文件类型plugin的脚本里. 你还可以加上你自己写的，请参考|write-filetype-plain|.

3. 使用缩进定义文件

   编辑程序的时候，语句的缩进可以让它自动完成. Vim为众多不同的文件类型提供了相应的缩进方案. 请参考|filetype-indent-on|和'indentexpr'选项.

ex command
```
autocmd FileType text setlocal textwidth=78
```

这让Vim可以自动断行，触发点是当前行已超过78个字符了. 但是只对类型是普通文本的文件生效. "autocmd FileType text"是一个自动命令. 它所定义的是每当文件类型被设置为"text"时就自动执行它后面的命令. "setlocal textwidth=78"把"textwidth"选项的值设置为78，但这种设置只对当前的一个文件有效.

ex command
```
autocmd BufReadPost *
    \ if line("'\"") > 0 && line("'\"") <= line("$") |
    \   exe "normal g`\"" |
    \ endif
```

这是另一个自动命令. 这次它服务的对象是所有类型的文件. 它所执行的复杂功能是检查是否定义了标记'"[2]，如果定义了就跳转到这个位

[1]译注: 或者是适合于你的个人爱好
[2]译注: 该标记记录了上次编辑一个文件时退出前光标的最后位置

置去. 每一行前面的反斜杠表示该行是前一行命令的延续. 它可以避免脚本中有些行变得过长. 请参考|line-continuation|. 这种表示法只能用于Vim脚本文件, 不要在冒号命令行上使用.

05.3 简单的映射

一个映射使你可以把一连串Vim命令以一个按键来表示. 假设说你要把某些词以花括号括起来. 或者说, 你需要把象"amount"这个的一个词变为"{amount}". 使用":map"命令, 你就可以告诉Vim按下F5就去完成这一操作:

ex command
```
:map <F5> i{<Esc>ea}<Esc>
```

> 备注: 练习这个命令时, 你必需键入4个字符来键入<F5>, 键入5个字符键入<Esc>, 而不是按下键盘上的F5键和ESC键. 在该手册中都要留心这种差异.

我们来做动作分解: <F5> F5功能键. 这是引起后面一连串命令开始执行的触发键.

i{<Esc> 插入{字符. <Esc>键结束当前的Insert模式.

e 将光标移动到当前word的最后一个字符上

a}<Esc> 在该word后面加上一个}字符.

定义了上述映射之后, 你要把一个word以花括号括起来就只需要把光标置于该word 的首字符上, 然后按F5.

本例中, 触发键是单个的按键; 但是映射本身可以是任意的一个字串. 这样一来如果你把一个Vim命令作为一个映射, 那这个命令就再也不具有它本身的意义了[1], 所以最好避免以一个Vim命令作为映射.

反斜杠也是一个可用于映射的字符. 你可能需要不止一个的映射, 比如说以"\p"来为一个word加上普通括号, 以"\c" 来加花括号. 就可以这样:

ex command
```
:map \p i(<Esc>ea)<Esc>
:map \c i{<Esc>ea}<Esc>
```

这样的映射对键入\与p之间的间隔有所要求, 快速键入才会使Vim把它们看作是一个映射而不是两个离散的普通字符[2]

[1]译注: 除非你删除这个映射, 此时该命令会恢复它的功能
[2]译注: 参考|timeoutlen|

":map"命令(不带参数)会列出当前已定义的映射. 它至少会包括定义在Normal 模式下的映射. 关于映射的更多内容请参考|40.1|.

05.4 添加一个plugin *add-plugin* *plugin*

Vim的功能可以通过向它添加plugin得以扩展. 所谓plugin不过是一个Vim会自动载入执行的脚本. 把一个脚本放入你的plugin目录就可以了, 非常容易.

plugin基本上分为两类:

全局的: 用于所有文件

专用于某类型文件的: 只用于特定类型的一类文件

下面先说全局的plugin, 接下来是专用于某种文件类型的plugin |add-filetype-plugin|.

全局的plugin

standard-plugin

当你启动Vim时, 它会自动载入一些全局的plugin, 你不必额外地做任何事情.这些plugin 定义了使用率很高的一些功能, 但它们是以一个Vim脚本的形式而不是通过内建于Vim可执行文件内来提供. 你可以在|standard-plugin-list|发现一个此类plugin 的列表. 另外请参考|load-plugins|.

add-global-plugin

你可以通过添加一个全局的plugin来获得额外的功能, 这只需两步: 1. 得到这个plugin文件. 2. 把它放到指定的目录下.

得到一个全局的plugin文件

在哪能找到?

1. 有一些随Vim一起发行. 位于目录$VIMRUNTIME/macros以及它的子目录
2. 从网上下载, 请查看

   http://vim.sf.net

3. 有时候也张贴在Vim的|maillist|.
4. 你也可以自己写: |write-plugin|.

使用一个全局的plugin

首先看一下这个plugin的内容，看看使用它要先满足什么条件．然后把它copy到你的plugin目录下：

List

系统	相应的plugin目录
Unix	~/.vim/plugin/
PC and OS/2	$HOME/vimfiles/plugin or $VIM/vimfiles/plugin
Amiga	s:vimfiles/plugin
Macintosh	$VIM:vimfiles:plugin
Mac OS X	~/.vim/plugin/
RISC-OS	Choices:vimfiles.plugin

以Unix为例(假设你还没有一个plugin目录)：

shell command

```
mkdir ~/.vim
mkdir ~/.vim/plugin
cp /usr/local/share/vim/vim60/macros/justify.vim ~/.vim/plugin
```

就这么简单！ 现在你就可以使用这些plugin里提供的新功能新命令了．

文件类型plugin

add-filetype-plugin *ftplugins* [1]

Vim的发布版中已经包括了针对不同文件类型的相应plugin，你可以使用下面命令开启对它的应用：

ex command

```
:filetype plugin on
```

就是这一个命令即可！ 另请查看|vimrc-filetype|．

如果你缺少某种文件类型的plugin，或者你找到一个更好的替代品，下面两个步骤可以增加一个文件类型的plugin：

1. 得到这个plugin．
2. 把它放到对应的目录里．

得到一个文件类型的plugin

与全局plugin所在的目录一样，通过查看这个plugin是否提到了某个文件类型，就可以知道它是全局的还是专用于某种文件类型的，在$VIMRUNTIME/macros下

[1] 译注： 可以看成是基于文件类型的插件

的脚本是全局的，而在$VIMRUNTIME/ftplugin目录下的则是专用于特定文件类型的.

使用一个文件类型plugin

ftplugin-name

你可以通过把一个文件类型plugin脚本放入相应的目录来完成对它的添加. 路径跟添加全局plugin时的一样，不过最后一个目录名是"ftplugin". 假设你找到了用于"stuff"文件类型的plugin，目前你在Unix系统上，那么你可以这样加入该文件：

shell command
```
mv thefile ~/.vim/ftplugin/stuff.vim
```

如果这个目录下已经有了一个同名文件. 你就要停下来仔细检查一下两个文件是否会引起冲突，如果没问题，你可以把要加入的新文件更名一下：

shell command
```
mv thefile ~/.vim/ftplugin/stuff_too.vim
```

下划线用于分隔用于标识文件类型的部分和其它部分，下划线其后的部分可以自由命名. 如果你用"otherstuff.vim"这样的名字，Vim可不能识别它，它只会在文件类型是"otherstuff"时被载入.

在MS-DOS上你不能使用长文件名. 如果你要用一个辅助的plugin但是文件类型字符串已经超过了6个字符[1]这就会有麻烦，不过你还可以通过一个额外的目录来处理这种情况：

shell command
```
mkdir $VIM/vimfiles/ftplugin/fortran
copy thefile $VIM/vimfiles/ftplugin/fortran/too.vim
```

文件类型plugin的文件名一般形式是：

List
```
ftplugin/<filetype>.vim
ftplugin/<filetype>_<name>.vim
ftplugin/<filetype>/<name>.vim
```

这里出现的"<name>"可以是任何你喜欢的名字. 以Unix系统下文件类型"stuff"为例：

[1]译注： MS-DOS文件名部分限制为小于等于8个字符，所以这里说文件类型字符串不能超过6个字符，因为下划线本身要占用一个字符，辅助plugin的其余部分至少会有一个字符

List
```
~/.vim/ftplugin/stuff.vim
~/.vim/ftplugin/stuff_def.vim
~/.vim/ftplugin/stuff/header.vim
```

<filetype>部分是plugin所服务的目标文件类型. 只有相应类型的文件才能应用到这个plugin. plugin文件中的<name>部分对Vim的识别工作并不起使用，你可以对几种不同的plugin都使用一样的<name>部分也没问题. 注意这些文件必需以".vim"为扩展名.

推荐阅读:

List

|filetype-plugins|	关于文件类型plugin的文档以及如何避免映射引起冲突的信息
|load-plugins|	关于Vim启动过程中何时载入全局plugin
|ftplugin-overrule|	如何强制改变全局plugin中的设置.
|write-plugin|	如何写一个plugin脚本.
|plugin-details|	关于如何使用plugin或者解决你的plugin出现的bug.
|new-filetype|	如何检测新文件类型

05.5 增加一个帮助文件 *add-local-help* *matchit-install*

幸运的话，你安装的plugin会附带一个帮助文件. 下面我们会解释如何安装一个帮助文件，这样你就可以很容易找到新的plugin的帮助.

我们使用"matchit.vim"这个plugin为例(这个脚本包括在Vim发行包中). 这个plugin使"%"命令可以跳转到匹配的HTML标签，也可以在Vim脚本中的if/else/endif关键字中跳转，非常好用，虽然它不兼容于Vim的早期版本(这正是为什么没把它作为一个默认功能的原因).

这个plugin有一个附带的文件: "matchit.txt". 我们首先把这个plugin复制到相应的目录. 这次我们不退出Vim来演示，所以我们可以使用$VIMRUNTIME这种形式的环境变量[1]. (如果你已经有了相应的目录就跳过那些"mkdir"命令)[2].

ex command
```
:!mkdir ~/.vim
:!mkdir ~/.vim/plugin
:!cp $VIMRUNTIME/macros/matchit.vim ~/.vim/plugin
```

[1]译注: 即使是在MS-DOS或MS-Windows下

[2]译注: MS-DOS或MS-Windows的用户可能奇怪为什么作者使用mkdir命令而不是md命令，答案是兼容性，mkdir可同时用于MS-DOS，MS-Windows 和Unix类系统，甚至Mac的最新操作系统OS X(它使用一个叫Darwin的Unix类内核)

现在在'runtimepath'选项里列出的目录列表中选一个目录，建立它的一个子目录"doc".

ex command
```
:!mkdir ~/.vim/doc
```

把帮助文件copy到这个"doc"目录下.

ex command
```
:!cp \$VIMRUNTIME/macros/matchit.txt ~/.vim/doc
```

奇迹发生了，现在你可以跳转到新的帮助文件中的帮助主题了：用|:helptags|命令生成一个本地化的的tags文件[1].

ex command
```
        :helptags ~/.vim/doc
```

现在你就可以帮助命令

ex command
```
        :help g%
```

来找到名为"g%"的帮助主题了.

ex command
```
        :help local-additions
```

来自本地化帮助文件的标题行自动被添加到该小节中.

通过该节列出的帮助主题你可以了解有哪些本地化的的帮助文件，并可以通过这些标签跳转到相应的帮助主题.

参考|write-local-help|了解更多关于写一个本地化的帮助文件的信息

05.6 选项窗口

如果你要查找一个选项，你可以在|options|帮助主题中寻找. 另外也可以使用这个命令：

[1]译注：tags文件发韧于程序员对编辑/浏览源程序的需要，比如在一个C/C++/Java源程序中，要跳转到某个名为foo的类的定义处，或是某个变量的声明，tags文件中的每一条目记录了这样一种程序元素在源代码中的位置，一些tags-aware的编辑器如Vim/Emacs可以根据tags文件中所记录的位置信息来快速跳转到目的地，tags所记录的位置信息一般以文件名+ 行号或文件名+搜索命令表达. 关于tags的更多信息，请参考man ctags，man etags

ex command
```
:options
```

该命令会打开一个新窗口，在该窗口的最开头的注释下面是一个选项列表，每行一个,对每个选项有一个对应的简短说明.这些选项根据主题分组． 把光标移动到你想了解的主题词上按下回车键就可以跳转到对该主题的详细解释． 再按下回车键或CTRL-O就又会回到该选项列表.

你还可以在此改变每个选项的设置．比如，移动到"displaying text"主题上，然后到下面这一行：

ex command
```
set wrap        nowrap
```

按下回车键，该行的内容就会变为：

ex command
```
set nowrap      wrap
```

该选项的值现在就被设置为关闭.

紧挨着该行之上是一个对'wrap'选项的简单描述． 把光标置于该行按下回车键会带你到'wrap'选项的详细解释去[1].

对于以一个数字或字符串为目标值的选项，你可以直接编辑选项的值，然后[2]按下回车键确认作出的修改并使之生效． 例如，把光标向上移动几行到：

ex command
```
set so=0
```

以"$"命令将光标置于字符0上． 用命令"r5"把它改为5． 然后按下回车键．现在你再四处活动光标时就会注意到光标快达到窗口的上下边界时周边文本的变化． 这就是'scrolloff'选项的结果，它决定了光标离窗口上下边界的最小行距为多少时会引起窗口滚动.

05.7 常用选项

Vim的选项可谓汗牛充栋．多数选项会被多数人冷落一旁． 其中一些常用的则人人青睐.本节也将特别照顾这些家常选项，不过别忘了你随时可以用":help"命令来获得关于它们的更详细解释，记住在选项关键字的前后放上一个单引号[3]，形如：

[1]译注： 不在当前缓冲区，在帮助文件options.txt中
[2]译注： 在Normal模式下
[3]译注： 这并不是必需的，只是为了最大限度地避免跳转到形近的帮助主题上去

ex command
```
:help 'wrap'
```

万一你把一个选项值改到不可收拾的境地，还可以在该选项的后面放一个&符号使它恢复其默认值，如：

ex command
```
:set iskeyword&
```

不要折行

Vim通常会把超出当前显示窗口显示宽度的行折到下一行显示，这样你还是可以看到整行的内容．有时候让它不管多长都放到窗口最右边去会更好．这时你要看这些超出当前视野的部分就要左右滚动该行了． 控制长行是否折到下一行显示的命令是：

ex command
```
:set nowrap
```

Vim会自动保证你把光标移动到某字符上时它会显示给你看，必要时它自动左右滚动.要查看左右10个字符的上下文[1]，用命令：

ex command
```
:set sidescroll=10
```

注意这只是改变文本的显示形式而不会影响内容本身．

跨行移动命令

Vim中多数移动光标的命令会在遇到行首或行尾时停止不动[2]． 'whichwrap'选项可以用来控制这些移动光标的命令此时的行为规则．下面的设置是它的默认值

ex command
```
        :set whichwrap=b,s
```

这样光标位于行首时按退格键会往回移动到上一行的行尾．同时在行尾按空格键也会移动到下一行的行首．

要让左右箭头键[3]在遇到行的边界时也能智能地上上下下，使用命令：

[1]译注： 也许叫左右文更合适些
[2]译注： 所谓畏首畏尾:-)
[3]译注： 对h，l命令无效

ex command
```
:set whichwrap=b,s,<,>
```

这些都是只针对于Normal模式． 要让左右箭头键在Insert模式下也能如此：

ex command
```
:set whichwrap=b,s,<,>,[,]
```

此外还有几个标志可以用于该选项，参考'whichwrap'．

查看制表符

文件中含有制表符时，你并不能看到它们．要让这些制表符成为可见的字符：

ex command
```
:set list
```

现在每个制表符都会以^I显示． 同时每行行尾会有一个$字符，以便你能一眼看出那些位于一行尾部的多余空格．

这样做的缺点是文件中制表符很多时整个屏幕看起来就很抱歉了．如果你的终端支持彩色显示，或者使用的是GUI，Vim就可以把制表符和空白字符高亮起来显示． 这要配合使用下面的'listchars'选项：

ex command
```
:set listchars=tab:>-,trail:-
```

现在每个制表符会以">---"显示[1]，同时行尾空格以"-"显示，看起来会好一点，你觉得呢?

关键字

'iskeyword'选项定义了一个word中可以包含哪些字符：

ex command
```
:set iskeyword
iskeyword=@,48-57,_,192-255 >
```

"@"在这里代指所有的字母． "48-57"指ASCII码从48到57的那些字符，即0到9． "192-255"是可打印拉丁字母．

有时候你可以想把连字符也视为word的一部分，这样象"w"命令就会把"upper-case"看作是一个word了：

[1]译注： 前提是仍然要:set list

ex command
```
:set iskeyword+=-
:set iskeyword
iskeyword=@,48-57,_,192-255,-
```

此时查看该选项的值的话你会发现Vim已经为新添加的成员同时配备了一个逗号以与其它成分分隔开来.

要把一个字符清理出去使用操作符"-=". 比如，要移走下划线:

ex command
```
:set iskeyword-=_
:set iskeyword
iskeyword=@,48-57,192-255,-
```

这次逗号又会被自动移走了.

信息显示区

Vim启动时会在窗口最底部留下一行用于显示信息. 要显示的信息太长时，Vim或者把它截短让你只能看到部分内容，或者多出来的信息需要你按下回车键以滚动显示.

你可以设置'cmdheight'选项来控制拿出几行来显示这些信息[1]. 比如:

ex command
```
:set cmdheight=3
```

当然这会让你的编辑区减少相应的行数，所以...，你自己拿主意吧.

下一章:	|usr_06.txt| Using syntax highlighting
版 权:	请参考|manual-copyright| vim:tw=78:ts=8:ft=help:norl:

[1]译注: 设置以后不显示信息的时候也会占据相应的空间

usr_06.txt Vim 6.3版.最后修改: 2002年07月14日

VIM用户手册--- 作者: Bram Moolenaar

使用语法高亮

黑纸白字了然无趣. 来点色彩才叫生活. 这不光是为了好看, 同时也会提高你的效率. 你为不同部分的指定不同的颜色. 也可以以屏幕上看到的颜色进行打印.

下一章: |usr_07.txt| 编辑多个文件
前一章: |usr_05.txt| 定制你的Vim
目　录: |usr_toc.txt|

06.1 打开色彩

简单的命令打开五彩斑斓的世界:

ex command
```
:syntax enable
```

多数情况下这会立即让你的文件增色生辉. Vim会自动检测到你的文件类型并为之载入相应的语法高亮. 突然之间注释变成了蓝色, 关键字是棕色, 字符串是红色. 整个文件的概况一目了然. 过一会之后你就会发现原来的黑白世界里真是白活了.

如果你想一直都用语法高亮, 可以把":syntax enable"命令放入你的|vimrc|文件.

如果你想只在终端支持彩色显示时才启用语法高亮, 可以在|vimrc|文件中这样设置:

ex command
```
if &t_Co > 1
   syntax enable
endif
```

如果你想只在GUI版本中使用语法高亮，只需把":syntax enable"放入|gvimrc|文件.

06.2 没有色彩或色彩错误?

看不到色彩可能是因为:

- 你的终端不支持彩色显示.

  Vim会用粗体，斜体和下划线来显示文本，但这看起来并不怎么样. 你可能会想用一个带有色彩支持的终端. 对Unix系统而言，我推荐XFree86项目的xterm: |xfree-xterm|.

- 你的终端是支持彩色显示，但是Vim不知道.

  确保$TERM变量设置正确. 比如，用的是xterm:

shell command
```
setenv TERM xterm-color
```

或(视你所用的shell而定):

shell command
```
TERM=xterm-color; export TERM
```

终端的名字必需与你实际所用的终端相符合. 如果还是不行，请参考|xterm-color|，此处提供了让Vim显示颜色的几个办法(不光是针对xterm).

- 不能识别文件类型

  Vim不可能识别所有的文件类型，有时侯几乎无法得知一个文件用的是什么语言. 试一下这个命令:

ex command
```
:set filetype
```

如果结果是"filetype="问题很可能就是Vim不知道文件类型. 你可以手工指定该文件的类型:

ex command
```
:set filetype=fortran
```

要知道一共就有哪些文件类型可用，请查看一下$VIMRUNTIME/syntax目录. 对GUI版本你还可以查看Syntax菜单. 也可以通过|modeline|设置文件类型，这样文件每次被编辑时都会被语法高亮. 比如，下面的这行可以放入Makefile 文件中(把它放在靠近文件结尾的地方):

ex command
```
# vim: syntax=make
```

你应该知道如何确定一个文件的类型. 通常来说是通过扩展名(文件名中.之后的部分). 请查看|new-filetype|了解Vim是如何确定一个文件的类型的.

- 你指定的文件类型没有语法高亮文件

    你可以手工设置它为一个相近的文件类型[1]. 如果看起来太过勉强，你也可以自己写一个语法高亮文件，请参考|mysyntaxfile|.

或者颜色有错:

- 被着色的文本读起来很费劲

    Vim会猜测你所使用的背景色. 如果背景是黑色的(或另一种比较暗的颜色)它就会用亮色来显示文字. 如果背景是白色(或另一种较亮的颜色)它就会暗色来显示文字. 如果Vim猜错了，很可能就会读起来很碍眼. 你可以设置'background'选项来改变对比度，比如使用暗色:

ex command
```
:set background=dark
```

使用亮色背景:

ex command
```
:set background=light
```

确保你把这行放在了":syntax enable"命令的前面，否则的话颜色已然被设置了就起不到作用了. 你可以在重新设置了'background'选项后用":syntax reset" 来让Vim重新设置默认颜色.

- 上下滚动时颜色有误

    Vim处理颜色时并不是通读整个文件进行解析. 它从你浏览的地方开始解析. 酱紫[2]会节省很多时间，但是有时候颜色就会弄错. 一个简单的办法是用CTRL-L. 或者稍往回滚动几行，请查看特定类型的语法高亮文件.比如Tex语法的|tex.vim|.

06.3 不同的颜色

如果你不喜欢默认的颜色，你可以选择另一种颜色方案. 在GUI中使用Edit/Color Scheme 菜单. 你也可以直接使用命令:

ex command
```
:colorscheme evening
```

"evening"是颜色方案的名字. 除此之外还有其它几种颜色方案. 请查看$VIMRUNTIME/colors目录

找到你钟爱的颜色方案后，可以在你的|vimrc|文件里加入":colorscheme"命令选择它.

你也可以写一个自己的颜色方案. 下面是实施步骤:

1. 找一个相近的颜色方案. 把该文件复制一份到你自己的Vim目录下. 对Unix系统

可以这样:

ex command
```
!mkdir ~/.vim/colors
!cp $VIMRUNTIME/colors/morning.vim ~/.vim/colors/mine.vim
```

这是在运行中的Vim中做的，因为它知道$VIMRUNTIME的值.

2. 编辑该文件. 下面的条目是十分有用的:

List

term	黑白终端的显示属性
cterm	彩色终端的显示属性
ctermfg	彩色终端的前景色
ctermbg	彩色终端的背景色
gui	GUI的显示属性
guifg	GUI的前景色
guibg	GUI的背景色

比如，要让注释变为绿色:

ex command
```
:highlight Comment ctermfg=green guifg=green
```

可以用于"cterm"和"gui"的属性是"bold"和"underline".
如果你想兼具两者的效果，可以写成"bold,underline".
更多的细节请参考|:highlight|.

3. 把下面这一行放入你的|vimrc|文件可以告诉Vim一直使用你自己的颜色方案:

ex command
```
colorscheme mine
```

如果你想看一下最常用的颜色设置都是什么样的效果,可以用下面的命令:

ex command
```
:edit $VIMRUNTIME/syntax/colortest.vim
:source %
```

你会看到几种不同的颜色组合. 检查一下哪一种看起来更好看可读性更好.

06.4 有色或无色

以彩色显示文本需要编辑器花额外的气力. 如果你发现显示变慢,你也可以暂时关闭语法高亮:

ex command
```
:syntax clear
```

编辑别的文件时(或者同一个文件也一样)又会应用彩色显示.

:syn-off

要彻底停用语法高亮可以用命令:

ex command
```
:syntax off
```

这将会彻底禁用语法高亮功能,并立即对各个缓冲区生效.

:syn-manual

如果你只想对某些文件施以语法高亮,用这个命令:

ex command
```
:syntax manual
```

这将会打开语法高亮功能,但并不在新开一个缓冲区时自动打开. 要为当前缓冲区打开语法高亮功能,可以通过这样设置'syntax'选项:

ex command
```
:set syntax=ON
```

06.5 彩色打印

在MS-Windows版本的Vim中你可以用下面的命令打印当前的文件:

ex command
```
:hardcopy
```

你会看到通常的打印对话框，在此可以选择一个目标打印机并进行相应的设置. 如果你有一个彩色打印机，打印结果应该跟你在Vim里看到的一样. 但如果你在Vim中应用的是暗色调的背景的话颜色会自动调整到适合在白纸上显示.

下面几个选项会影响到Vim中的打印:

List
```
'printdevice'
'printheader'
'printfont'
'printoptions'
```

要打印部分行，可以使用Visual模式选择被打印行，用下列命令:

normal mode command
```
v100j:hardcopy
```

"v"命令进入Visual模式. "100j"向下移动100行，这些被选取的行将以高亮显示. 接下来的":hardcopy"将打印这些行. 当然你也可以用其它的位移命令来在Visual 模式中进行选取.

在Unix系统上也一样可以，如果你有一个兼容PostScript的打印机，直接这样做就可以. 否则的话，稍微麻烦一点. 你得把文件先转换到HTML格式，然后从Netscape这样的浏览器中打开该HTML文件进行打印.

把当前文件转到HTML格式使用这个命令:

ex command
```
:source $VIMRUNTIME/syntax/2html.vim
```

运行后你会看到它忙个不停，文件太大时这可要花些时间. 完成后会在另一个窗口中显示生成的HTML代码. 现在你可以把它保存起来了(存到哪并不重要，反正用完后你就可以丢掉它了):

ex command
```
:write main.c.html
```

在你喜欢的浏览器中打开该文件就可以打印它了. 一切顺利的话, 输出结果应该跟在Vim 里看到的一样. 请参考|2html.vim|了解更多. 别忘了打印机完后把刚才的HTML文件删掉.

除了用来打印, 你也可以把生成的HTML文件放在一个Web服务器上, 这样其它人就可以看到你那漂亮的带语法高亮的代码了.

06.6 进一步的学习

|usr_44.txt| 自定义语法高亮文件|syntax| 囊括所有细节.

下一章: |usr_07.txt| 编辑多个文件
版 权: 请参考|manual-copyright| vim:tw=78:ts=8:ft=help:norl:

usr_07.txt Vim 6.3版.最后修改: 2004年03月12日

VIM用户手册--- 作者: Bram Moolenaar

编辑多个文件

不管你有多少文件要编辑，你都可以在Vim在处理它们. 可以定义一个要编辑的文件列表. 从一个文件转到另一个文件. 也可以在不同文件之间复制粘贴.

下一章: |usr_08.txt| 分隔窗口
前一章: |usr_06.txt| 使用语法高亮
目 录: |usr_toc.txt|

07.1 编辑另一个文件

目前为止我们使用Vim的方式还是为每一个要编辑的文件运行一次Vim. 这是最简单的用法. 命令

ex command
```
:edit foo.txt
```

可以在当前Vim中开始编辑另一个文件. 当然你可以用任何文件名来替代"foo.txt". Vim会关闭当前正在编辑的文件打开指定的新文件进行编辑. 如果当前文件还有未存盘的内容，Vim会显示如下的错误消息同时也不会打开另一个文件:

Display
```
E37: No write since last change (use ! to override)
```

备注: Vim在每条错误消息前放上它对应的错误ID号，这样如果你从简单的错误信息中还不知道错误的原因时，可以通过帮助系统查找这个ID: :help E37

此时，你可以有几种选择．你可以保存该文件：

ex command
```
:write
```

或者你可以强制Vim丢弃当前未保存的修改并开始编辑新的文件，使用强制执行修饰符：

ex command
```
:edit! foo.txt
```

如果你想编辑另一个文件，但又不想保存当前文件中的改动[1]，你可以使它变为一个隐藏的缓冲区：

ex command
```
:hide edit foo.txt
```

被修改过的文本还在，只是你看不到它而已． 在|22.4|中讲解缓冲区列表的主题中对此有详细的解释

07.2 文件列表

你可以在启动Vim时就指定要编辑多个文件．如：

shell command
```
vim one.c two.c three.c
```

该命令启动Vim并告诉它你要编辑3个文件．Vim将在启动后只显示第一个文件．完成该文件的编辑后，可以以命令：

ex command
```
:next
```

开始下一个文件的编辑.如果你的当前文件中有未存盘的内容，你会象前面一样得到一个错误消息，":next"命令也不会继续．这与前面提到的":edit"命令一样． 要放弃这些改动，用：

ex command
```
:next!
```

但多数情况下人们还是要保存工作成果并继续下一个文件的编辑．有一个命令合并了这个过程：

[1]译：当然也不想放弃这些改动

ex command
```
:wnext
```

这个命令完成以下两个单独命令的工作[1]:

ex command
```
:write
:next
```

当前在编辑哪个文件?

可以通过查看窗口的标题条得知你当前正在编辑的文件名. 应该也会同时显示出象"(2 of 3)"这样的信息. 这意味着你正是编辑一个由3个文件组成的文件列表中的第2个.

如果你想查看整个列表中就有哪些文件, 使用命令:

ex command
```
:args
```

这是"arguments"的简写形式. 输出结果形如:

Display
```
one.c [two.c] three.c
```

这就是你启动Vim时指定的要编辑的文件列表. 你当前正在编辑的那一个文件将会以方括号括起来.

移到另一个文件

要回到前一个文件:

ex command
```
:previous
```

就跟":next"命令一样, 不过它是朝向另一个文件. 同样有一个对应的快捷方式命令:

ex command
```
:wprevious
```

要移到最后一个文件:

[1]译: 规律, 对于使用频率极高的命令序列, Vim会提供一个单一的命令来做本可以由几个命令组合起来完成的操作. 但你不能这样任意组合基本命令

ex command
```
:last
```

到第一个:

ex command
```
:first
```

不过没有":wlast"或者":wfirst"这样的命令.

你也可以在":next"和":previous"命令前面使用一个命令计数. 要向前跳过2个文件:

ex command
```
:2next
```

自动存盘[1]

当你在不同文件之间转移时，你必需记住用":write"命令来存盘. 否则就会得到一个错误消息. 如果你确定自己每次都是要保存文件，就可以告诉Vim每当需要时就自动保存文件，不必过问:

ex command
```
:set autowrite
```

如果你正在编辑一个不希望它被自动保存的文件，还可以把该选项关闭:

ex command
```
:set noautowrite
```

编辑另一个文件列表

不用重新启动Vim，你就可以重新定义一个文件列表. 下面的命令定义了要编辑3个文件:

ex command
```
:args five.c six.c seven.h
```

或者用一个通配符，就象在shell使用通配符一样:

ex command
```
:args *.txt
```

[1]译: 这里说的自动存盘指的是某个事件发生时自动保存文件，而不是象word中每隔一段时间就自动保存一次文件，当然Vim中也有此功能，但此处另有所指

Vim会打开列表中的第一个文件. 同样, 如果当前文件被改动但没有存盘, 你需要先保存当前的文件, 或者用":args!"(加了一个!)放弃当前文件中未存盘的内容.

你编辑过最后一个文件了吗?

arglist-quit

当你有一个列表的文件要编辑时, Vim假设你要全部编辑它们. 为防止你漏掉某些文件过早地退出, Vim会在你没有编辑过最后一个文件就想退出时给出一个错误信息:

Display
```
E173: 46 more files to edit
```

如果你确定要退出, 只要再执行一次退出命令. 这次可以真正退出了(但是不要在这两次执行同样的命令中间再做其它操作)

07.3 切换到另一文件

要在两个文件间快速切换, 使用CTRL-^(在美语键盘上^位于字母键区的6上). 如:

ex command
```
:args one.c two.c three.c
```

当前编辑的文件是one.c.

ex command
```
:next
```

现在变成two.c了. 使用CTRL-^可以让你再回到one.c. 再执行一次CTRL-^又会再转到two.c 如此循环, 如果你执行的是:

ex command
```
:next
```

现在你会转到three.c. 注意CTRL-^命令并不改变当前你在文件列表中的位置, 只有命令":next"和":previous"才会引起此位置的变化.

你上一个编辑的文件叫"alternate"文件. 所以刚进入Vim时就用这个命令的话它就无事可做, 因为你还没有编辑过任何其它的文件.

预定义的标记

跳转到另一个文件后, 你还可以使用两个十分有用的标记:

normal mode command
```
`"
```

这个标记会带你到上次你离开该文件时光标所在的位置. 另一个标记则是你最后一次对文件做出改动处:

normal mode command
```
`.
```

假设你正在编辑的是"one.txt". 在文件半中间的某个地方你用"x"命令删除了一个字符.然后你用"G"命令到了最后一行, 用":w"命令保存该文件后转而编辑其它几个文件, 最后又用":edit one.txt"回到该文件. 如果现在你用`"命令Vim就会跳转到该文件的最后一行, 那是上一次你关闭该文件时的光标位置. 使用`.则带你到你用"x"删除了一个字符的地方. 即使你已经在该文件来回移动了多次, `"和`.这两个标记还是忠实是记录着这两个特殊的位置. 除非你又一次对该文件做出改动或关闭该文件.

文件标记

第4章中我们说过可以用"mx"在一个中某个设置一个标记, 然后用"`x"可以将光标移到该位置. 这只在当前文件内有效, 如果你编辑了其它的文件并且也在其中设置了标记, 这些标记将只对这个的文件有效. 每个文件都有它自己的标记. 它们是局部于文件的.

目前为止我们用的标记还都是以小写字母命名的. 还有一种以大写字母命名的标记. 它们是全局标记, 它们可以用在任何文件中. 比如假设我们正编辑"foo.txt". 到文件的半中间("50%")处设置一个名为F的标记(F意为foo):

normal mode command
```
50%mF
```

现在转而编辑"bar.txt"并在其最后一行设置一个名为B(B意为bar)的标记:

normal mode command
```
GmB
```

现在你可以用"´F"命令跳转到文件foo.txt的半中间. 或者编辑另一个文件, "´B"命令会再把你带回文件bar.txt的最后一行.

Vim会一直记得你在文件中设置的标记, 直到你改变标记的位置为止. 所以你可以设置一个标记后成几个小时做别的事情, 需要的时候还可以用该标记回到它所代表的位置[1]

[1]译注: 如果你删除了标记所在的行, 同时也就等于删除了该标记

把标记的名字与它所代表的位置联系起来会十分好记. 比如, 用H代表header文件, M 代表Makefile, C代表C源文件.

要知道某个标记所代表的位置是什么, 可以将该标记的名字作为"marks"命令的参数:

ex command
```
:marks M
```

你也可以连续跟上几个参数:

ex command
```
:marks MCP
```

别忘了你还可以用CTRL-O和CTRL-I可以跳转到较早的位置和靠后的某位置.

07.4 备份

通常情况下Vim不会生成备份文件. 如果你需要的话, 只需要执行命令:

ex command
```
:set backup
```

生成的备份文件名将是原文件名后面附加一个~. 如果原文件是data.txt, 则生成的备份文件是data.txt~.

如果你不喜欢这个默认的备份文件名后辍, 你可以用下面的命令重新指定一个:

ex command
```
:set backupext=.bak
```

这将会生成一个名为data.txt.bak的备份文件.

另一个与此有关的选项是'backupdir'. 它指定了备份文件将被置于哪个目录下. 默认是写到原文件所在的目录下. 多数情况下人们需要的正是这样.

> 备注: 如果'backup'选项是关闭的但'writebackup'选项是打开的, Vim还会生成一个备份文件. 但是, 一旦该文件被成功地保存它就会被自动删除. 如果因为某种原因(比如磁盘满或被雷电击中, 虽然后者不常发生)原文件不能保存.这倒不失为一种保护文件的办法,

保存原始版本

如果你在编辑的是程序源文件，你可能会希望保存一份修改前的原始文件的一个副本． 但是用备份文件的话它会在每次你写文件时被覆盖．这样备份文件将总是保存前一个版本的内容，而不是原始的版本．

'patchmode'选项可以让Vim保存原始文件，它指定了备份该原始版所用的文件扩展名：

ex command
```
:set patchmode=.orig
```

如果你第一次开始编辑data.txt文件，改一些东西然后存盘，Vim会保留一份该文件的原始版在"data.txt.orig"中．

如果你继续修改该文件，Vim也会注意到名为"data.txt.orig"的文件已经存在，后续生成的备份文件将被命名为"data.txt~"(或者你用'backupext'选项指定的其它扩展名)．

如果你把'patchmode'选项设置为空(默认情况正是如此)，文件的原始副本就不会被额外保存．

07.5 在文件间复制粘贴

本节讲述如何在不同文件之间复制内容，我们以一个简单的例子开始． 首先编辑你希望从中复制内容的文件．将光标移到某处文件并按下"v"． 该命令将开始Visual模式． 现在把光标移到要复制文件的末尾按下"y"． 该命令将yanks(复制)被选择的内容．

要复制上面这一段的话，你要做的是[1]：

ex command
```
:edit thisfile
/This
vjjjj$y
```

现在开始编辑你希望把复制的内容放入其中的文件．把光标置于你希望复制内容的地方，用"p"把此前复制的内容粘贴到这里．

ex command
```
:edit otherfile
/There
p
```

[1]译：因为以文档本身为例，所以这里的说法只对英文版帮助文件有意义

当然你可以用其它的命令来yank要复制的内容. 比如用"V"命令进入Visual模式整行整行地选择文本. 或者用CTRL-V来选择一个矩形块的内容. 或者用"Y"选择当前行单行的内容, 用"yaw"来yank-a-word, 等等.

"p"命令将把复制的内容放到光标之后. "P"则可以把要复制的内容放在光标之前. 注意Vim会知道你复制的内容是整行的内容还是一个矩形块, 粘贴这些内容时也会采用相应的方式.

使用寄存器

如果你要从一个文件中复制出好几块独立的文本到另一个文件中去, 单用上面的方法就不得不多次切换文件, 存盘. 将这些独立的文本存到一个寄存器中去可以避免这种繁琐的切换.

一个寄存器只是Vim用来存放文本的地方. 这里我们只用从a到z这26个字母作为寄存器的名字(稍后你会发现还有其它可用的寄存器). 来把一个句子复制到名为f的寄存器中(f 意为first):

normal mode command
```
"fyas
```

"yas"命令象前面一样复制一个句子. 告诉Vim把复制的内容放到寄存器f中的部分是"f. 而且必需放在复制命令的前面. 现在把3个整行的内容放到寄存器l中(l 意指line):

normal mode command
```
"l3Y
```

命令计数也可以放在"l的前面. 要复制一个文本块到寄存器b中(b意为block):

normal mode command
```
CTRL-Vjjww"by
```

注意指定寄存器的部分"b紧挨在y命令的前面. 这是必需的. 放在w"命令前面就不行.

现在你分别在寄存器f, l, 和b中保存了3块不同的内容. 开始另一个文件的编辑, 将光标移到你想复制内容的地方然后:

normal mode command
```
"fp
```

指定寄存器的部分"f必需出现在p命令的前面.

你可以以任何顺序复制这3个寄存器中的内容. 其中的内容也会一直保存, 直到你再次使用该寄存器保存内容时覆盖了它[1]. 这样你可以多次复制其中的内容.

删除内容时, 也可以指定一个寄存器名. 这种办法可以用来移动多处的文本. 比如, 下面的命令删除了一个word并把它保存在名为w的寄存器中:

normal mode command
```
"wdaw
```

指定寄存器名的部分又一次出现在删除命令"d"的前面.

向文件中追加内容

要把多行文本收集到一起写入一个文件, 可以用命令:

ex command
```
:write >> logfile
```

这将会把当前文件的内容追加到文件"logfile". 这样做避免了你使用前面的方法去复制内容, 编辑log文件. 这样可以省去两个环节. 但它的局限是只能在文件的最后追加内容.

要想只追加几行的内容到文件中去, 可以在使用命令":write"之前先在Visual模式下选定要写入的内容. 在第10章你会了解其它选择文本行的办法.

07.6 查看文件

有时候你只想查看文件的内容而已, 并不会向其中写入什么东西. 但不假思索就用":w"可会招致覆盖原始文件的风险. 要避免这种错误, 可以以只读方式编辑文件.

下面的命令以只读方式运行Vim:

shell command
```
vim -R file
```

在Unix上下面的命令是等价的[2]:

shell command
```
view file
```

[1]译: 还可以向寄存器中追加内容而不覆盖先前的内容
[2]译: 可以用批处理在MS-Windows或MS-DOS上做一个等价的命令

现在你将在只读模式编辑"file"，此时尝试用":w"会招来一个错误消息告诉你该文件不能被保存.

如果你尝试修改这个文件的话也会得到一个警告消息:

Display
```
W10: Warning: Changing a readonly file
```

不过你还是可以做出修改. 这样人们可以为了方便浏览起见格式化该文件.

如果你改动了该文件但忘了它是只读的，你还是可以保存该文件. 在":write"命令之后使用!强制执行修饰符[1].

如果是想强制性地避免对文件进行修改，可以用命令:

shell command
```
vim -M file
```

这样每个修改文件的尝试都会失败. 帮助文件就是这样，比如，你试着去修改帮助文件时会看到这样的错误信息:

Display
```
E21: Cannot make changes, 'modifiable' is off
```

你可以用-M选项告诉Vim工作在viewer模式. 这都是表明你自愿如此，毕竟下面的命令还是可以去掉这层保护:

ex command
```
:set modifiable
:set write
```

07.7 更改文件名

编辑一个新文件的最好办法是以一个内容相似的文件为基础进行修改. 比如，你要写一个移动文件的程序. 同时你已经有了一个可以复制文件的程序，你可以这样开始:

ex command
```
:edit copy.c
```

你可以删除其中不需要的部分. 现在可以把它存成一个新文件了. ":saveas" 命令正好可堪此任:

[1]译: 此处的!是强制保存修改而不是丢弃修改的内容

ex command
```
:saveas move.c
```

Vim会以给定的文件名保存当前缓冲区中的内容，同时开始编辑该文件．这样下次你再用":write"命令的话，它就是存成"move.c"，而"copy.c"还保持原来的内容．

如果你想改变当前正在编辑的文件名，但不想保存该文件，就可以用命令：

ex command
```
:file move.c
```

Vim将把该文件标记为"not edited"[1]． 这意味着Vim知道这不是你进入Vim时开始编辑的文件．你保存该文件时就会收到这样的错误信息：

Display
```
E13: File exists (use ! to override)
```

这可以保护你意外地覆盖了其它文件．

下一章：|usr_08.txt| 分隔窗口

[1]译：中文版为"未编辑"

usr_08.txt Vim 6.3版.最后修改: 2004年06月04日

VIM用户手册--- 作者: Bram Moolenaar

分隔窗口

同时显示两个不同的文件，或者同时查看同一个文件的两个不同位置，或者是同步显示两个文件的不同之处. 所有这些都可以通过分隔窗口的功能来实现.

下一章: |usr_09.txt| 使用GUI
前一章: |usr_07.txt| 编辑多个文件
目　录: |usr_toc.txt|

08.1 分隔一个窗口

打开一个新窗口最简单的办法就是使用命令:

ex command
```
:split
```

该命令将屏幕分为上下两个窗口并将光标定位在上面的窗口中:

Display
```
+----------------------------------+
|/* file one.c */                  |
|~                                 |
|~                                 |
|one.c=============================|
|/* file one.c */                  |
|~                                 |
|one.c=============================|
|                                  |
+----------------------------------+
```

你看到的是两个窗口，显示的内容却来自同一个文件. 含有"===="的行表示状态行. 它显示了关于它上面的窗口的相关信息. (实际情况下状态行会反相显示)

同时打开两个窗口可以让你查看同一文件的两个不同部分. 比如你可以让上面的窗口来显示一个程序中的变量声明部分，下面的窗口是使用了这些变量的编码区.

CTRL-W w命令可以切换当前活动窗口. 如果你在上面窗口，它会把它带到下面. 如果你在下面的窗口，同样的命令却是把你带到上面. (CTRL-W CTRL-W功能相同，只不过你可以迟一点松开CTRL键)

关闭窗口

命令

ex command
```
:close
```

可以关闭当前窗口. 实际上，任何退出文件编辑的命令象":quit"和"ZZ"都会关闭窗口，但是用":close" 可以阻止你关闭最后一个Vim，以免以意外地整个关闭了Vim.

关闭除当前窗口外的所有其它窗口

如果你打开了一大堆窗口，但现在你只想把重心放在其中一个上面，这时命令

ex command
```
:only
```

就十分有用了. 它会关闭除当前窗口外的所有其它窗口. 如果这些窗口中有被修改过的，你会得到一个错误信息[1]，同时那个窗口会被留下来.

08.2 为另一个文件分隔出一个窗口

下面的命令可以打开第二个窗口同时在新打开的窗口中开始编辑作为参数的文件:

ex command
```
:split two.c
```

[1]译: 在支持中文的gvim上显示的信息是"E445:其它窗口有改变的内容"

如果你目前正编辑的文件名为one.c，那么执行该命令后的屏幕大致象这样：

Display
```
+----------------------------------+
|/* file two.c */                  |
|~                                 |
|~                                 |
|two.c=============================|
|/* file one.c */                  |
|~                                 |
|one.c=============================|
|                                  |
+----------------------------------+
```

如果要打开一个新窗口并开始编辑一个空的缓冲区，使用命令：

ex command
```
:new
```

你可以重复使用":split"和":new"命令打开任何你喜欢的窗口数目[1]

08.3 窗口大小

":split"命令还可以接受一个参数. 如果指定了这个参数的话，它将会作为新打开窗口的高度. 比如下面的命令就打开了一个高度为3行的新窗口并在其中编辑名为alpha.c的文件[2]：

ex command
```
:3split alpha.c
```

对于已经打开的窗口有好几种办法可以改变它们的大小. 如果你还有鼠标可用的话就更容易了： 把鼠标移到分隔窗口的状态行上，上下拖动它即可.

增加当前窗口高度：

normal mode command
```
CTRL-W +
```

减小：

[1]译：但在一个17英寸的显示器上，你不会喜欢5个以上同时打开的窗口
[2]译：这里的3可不是命令记数，而是命令的参数

normal mode command

```
CTRL-W -
```

这两个命令都可以接受一个命令记数，用以一次将窗口的高度增减指定的行数． "4 CTRL-W +"将使当前窗口增加4行高度．

将窗口高度指定为一个固定的高度：

normal mode command

```
{height}CTRL-W _
```

这个命令的组成是： 一个代表行数的数字{height}，CTRL-W和一个下划线(在标准键盘上同时按下Shift键和-键)．

要让窗口达到它可能的最大高度，不指定命令记数直接使用CTRL-W _．[1]

使用鼠标

在Vim中大多数工作都可以通过键盘有效地完成． 不幸的是调整窗口大小的命令需要敲太多的键．这时用鼠标反而更快． 将鼠标置于状态行．按下鼠标左键拖动．状态行就会跟着上下移动，相应地窗口的高度也跟着变大变小．

相关选项

'winheight'选项可以设置为一个你期望的最小的窗口高度． 'winminheight'则用于设置一个强制的最小高度

同样地，有一对对应的选项:'winwidth'和'winminwidth'，分别用于指定期望的最小窗口宽度和强制的最小窗口宽度．

如果设置了'equalalways'选项，则Vim在每次打开或关闭窗口之际都会自动让所有窗口均摊屏幕上可用的高度和宽度．

08.4 垂直分隔

":split"命令创建的新窗口位于当前窗口之上． 要让新窗口出现在当前窗口的左边，可以用命令：

ex command

```
        :vsplit
或:
        :vsplit two.c
```

[1]译：可以记为让窗口W自己CTRL控制高度，达到它的最大底线_

分隔后的窗口大致象:

Display
```
+----------------------------------------+
|/* file two.c */    |/* file one.c */   |
|~                   |~                  |
|~                   |~                  |
|~                   |~                  |
|two.c===============one.c=============|
|                                        |
+----------------------------------------+
```

实际操作时这里窗口中间出现的|都会以反相显示. 这叫垂直分隔符. 它用来界定左右两个窗口.

同样有一个对应的":vnew"命令,用于垂直分隔窗口并在其中打开一个新的空缓冲区. 与此等价的一个命令是:

ex command
```
:vertical new
```

实际上":vertical"可以出现在任何分隔窗口的命令前[1]. 这将使接下来的窗口分隔命令进行垂直方向的分隔而不是水平方向上. (如果随后的命令跟分隔窗口无关,这个前缀就形同虚设).

切换窗口

因为你可以以水平和垂直方向任意分隔窗口,最终的窗口布局也会五花八门. 置身于众多的窗口你需要在里面来去自如:

Display

CTRL-W h	到左边的窗口
CTRL-W j	到下面的窗口
CTRL-W k	到上面的窗口
CTRL-W l	到右边的窗口
CTRL-W t	到顶部窗口
CTRL-W b	到底部窗口

有没有觉得这些字符有些眼熟[2]. 如果你愿意的话,用光标键来也同样可以.

参考|Q_w|可以了解更多的关于在窗口间移动的命令.

[1]译注: 事实上它也可以出现在跟分隔窗口无关的命令前,如:vertical echo "hello",只不过没有效果罢了

[2] 译注: hljk是左右上下以字符为单位移动, CTRL-W hljk则是以窗口(W)为控制单位移动

08.5 移动窗口

如果你已经分隔出了几个窗口，但对它们的位置不满意．这时你需要一个命令来移动它们的相对位置．比如说，你已经有了下面三个窗口：

Display
```
+----------------------------------+
|/* file two.c */                  |
|~                                 |
|~                                 |
|two.c=============================|
|/* file three.c */                |
|~                                 |
|~                                 |
|three.c===========================|
|/* file one.c */                  |
|~                                 |
|one.c=============================|
|                                  |
+----------------------------------+
```

显然最后一个窗口本应在最上面．转到该窗口(使用CTRL-W w)然后键入如下命令：

normal mode command
```
CTRL-W K
```

这里使用的是大写的字母K．命令的结果是将当前窗口向上提升了一次．有没有注意到K又被用于向上移动[1]．

如果你已经有了几个垂直分隔的窗口，CTRL-W K会把当前窗口向上移动同时占据整个Vim程序窗口的宽度．假如当前的窗口布局是：

Display
```
+----------------------------------------------+
|/* two.c */   |/* three.c */   |/* one.c */   |
|~             |~               |~             |
|~             |~               |~             |
|~             |~               |~             |
|~             |~               |~             |
|~             |~               |~             |
|two.c=========three.c=========one.c=========|
|                                              |
+----------------------------------------------+
```

[1]译：看得出来，作者对这种命令字符的精心选取颇为得意

对中间的那个窗口(three.c)应用命令CTRL-W K将使窗口布局改为[1]:

Display

```
+----------------------------------------+
|/* three.c */                           |
|~                                       |
|~                                       |
|three.c=================================|
|/* two.c */            |/* one.c */     |
|~                      |~               |
|two.c==================one.c===========|
|                                        |
+----------------------------------------+
```

另外三个相似的命令是(估计你已经猜到了):

Display

```
CTRL-W H        向左移动窗口
CTRL-W J        向下移动窗口
CTRL-W L        向右移动窗口
```

[2].

08.6 针对所有窗口操作的命令

在打开一大堆窗口的情况下要退出Vim, 你可以一个一个地关闭这些窗口. 还有另外一个专用的命令:

ex command

```
:qall
```

意思很明显"quit all"[3]. 如果这些窗口中有被修改又没保存的, Vim 就不会退出. 光标也会自动被定位到该窗口中. 这样你可以用":write"来保存修改, 或用":quit!"放弃这些改动.

如果你已经知道有窗口被修改了而且还没有保存, 可以用命令

ex command

```
:wall
```

[1]译: 对一个水平分隔的上下两个窗口使用CTRL-W L 试试, 你会发现这一命令的副作用还可以把水平分隔的窗口变为垂直分隔

[2]译: 对于相似的命令, 本书一贯的作法是以其中之一为例详细讲解, 其余的则一笔代过, 留给读者去举一反三

[3]译: 退出全部窗口

来保存所有被修改的窗口． 命令意为"write all"[1]． 但实际上，它只会存盘那些改动过的． Vim很清楚重写一遍完全没有改变的文件毫无意义．

还有一个对":qall"和":wall"的组合：保存并退出所有窗口：

ex command

```
:wqall
```

这个命令将保存所有被修改的文件然后退出Vim．

最后，还有一个放弃所有修改强制退出Vim的命令：

ex command

```
:qall!
```

慎用！ 这一丢可就再也回不来了！

为每一个文件打开一个窗口

使用"-o"选项可以让Vim为每一个文件打开一个窗口：

shell command

```
vim -o one.txt two.txt three.txt
```

结果是：

Display

```
+-------------------------------+
|file one.txt                   |
|~                              |
|one.txt========================|
|file two.txt                   |
|~                              |
|two.txt========================|
|file three.txt                 |
|~                              |
|three.txt======================|
|                               |
+-------------------------------+
```

"-O"参数可以使打开的窗口都垂直排列．

如果已经进入了vim， ":all"命令会为命令行上指定的所有文件各开一个窗口． ":vertical all"则让打开的窗口都是垂直分隔．

[1]译：存盘所有的缓冲区

08.7 使用vimdiff查看不同

Vim有一种特殊的启动方式，可以显示两个文件的不同之处． 我们来以"main.c"文件为例，在其中一行插入几个字符，在打开'backup'选项的情况下保存文件，这样名为"main.c~"的备份文件会保留该文件此前的版本．

在一个shell中键入如下命令(注意不是在Vim中)：

shell command
```
        vimdiff main.c~ main.c
```

Vim将会打开左右两个垂直分隔的窗口． 你会只看到你多插入了几个字符的那行以及它周围的上下几行内容．

Display
```
 VV                   VV
+-----------------------------------------+
|+ +--123 lines: /* a|+ +--123 lines: /* a|   <- fold
|  text              |  text              |
|  text              |  text              |
|  text              |  text              |
|  text              |  changed text      |   <- changed line
|  text              |  text              |
|  text              |  ------------------|   <- deleted line
|  text              |  text              |
|  text              |  text              |
|  text              |  text              |
|+ +--432 lines: text|+ +--432 lines: text|   <- fold
|  ~                 |  ~                 |
|  ~                 |  ~                 |
|main.c~==============main.c==============|
|                                         |
+-----------------------------------------+
```

(该图没有显示语法高亮，实地使用vimdiff命令会更好看一些)

没被修改的行被缩置到单独的一行中． 这叫折叠．被执行的行以"<-fold"为标识． 单个的折叠行代表了123行的内容．这些行对于两个文件来说都是一样的．

以"<- changed"标识的那一行以高亮显示，被插入的字符也以另类的颜色突出显示． 这种方式清晰地展示了两个文件的异同之处．

在man.c窗口中显示为"---"行表明该行被删除了． 如图中标以"<-delete line"的行． 注意这些字符并不真的是文件内容的一部分，它们

只是被用来填满main.c的空缺部分，这样两个文件就可以显示相同的行数了.

折叠的栏位

两个比较窗口的左边都有一个背景略有不同的栏位. 上图中它们以"VV"标识. 看到折叠行前面的加号字符了吗. 把鼠标移到该字符上单击. 折叠的行就会展开，这样你就可以看到被隐藏起来的内容了.

折叠栏前面的减号表明这是已经被打开了折叠行. 单击该符号会再次折行. 当然，你要有鼠标可用才行. 如果没有，也可以用使用"zo"来展开折叠，用"zc"再把它们折起.

运行VIM后比较不同

另一种进入diff模式的办法可以在Vim运行中操作. 编辑文件"main.c"，然后打开另一个分隔窗口显示其不同[1]:

ex command
```
:edit main.c
:vertical diffsplit main.c
```

":vertical"命令让打开的对比窗口以垂直方向分隔. 如果没有它，打开的窗口就是水平方向分隔的.

如果你有一个patch或diff文件，还有第3种开始diff模式的方法. 首先编辑那个要应用patch 的文件. 然后告诉Vim patch文件的名字:

ex command
```
:edit main.c
:vertical diffpatch main.c.diff
```

警告: patch文件必需只包含了单个文件的patch才行. 否则你会看到一大堆错误信息，同时也有可能把文件打上错误的补丁.

补丁会打到当前文件的一个副本上，该文件本身并不会被修改(除非你决定以打完补丁后的内容保存它).

同步滚动

如果两个文件有很多不同之处，你可以以通常方式滚动窗口进行查看. Vim会保证两个窗口总是显示文件中相同位置[2]的内容，所以你可以一行对一行地看到它们的差异.

如果暂时不想让它这样，使用命令:

[1]译注: 以diff模式后如何退出到普通模式??
[2]译: 指行号相同的行

```
ex command
:set noscrollbind
```

即可

跳到不同之处

如果你禁用了折行显示，要找到两个文件的不同之处就要费劲些，命令：

```
normal mode command
]c
```

可以直接向前定位到下一个不同之处． 向后定义下一个发生改变的行用：

```
normal mode command
[c
```

以一个数字为命令记数可以加快跳转的步伐．

消除差异

你可以在两个对比窗口中移动文字． 这样做会引起两个文件对比结果的变化． 不同之处会减少或增多． Vim并不时时更新对应的高亮显示．命令：

```
ex command
:diffupdate
```

可以在需要的时候重新比较两个文件．要消除一个不同之处，你可以把高亮起来的文件从一个窗口移到另一个窗口去． 以上面的"main.c"和"main.c~"为例． 把光标置于左边窗口中比右边窗口多出的一行上． 现在使用命令：

```
normal mode command
dp
```

两个文件的不同被消除了，当前窗口中引起不同的内容被放到另一窗口中缺少这段内容的地方去了． "dp"是"diff put"的缩写．

也可以用其它方法来做．将光标移到右边的窗口，到"changed"插入的位置．

键入命令：

```
normal mode command
do
```

现在也消除了该位置的不同之处，Vim从另一窗口中的对应位置取来了差异的内容．因为现在两个文件完全相同了，Vim将把所有内容都折叠起来． "do"意为"diff obtain"[1]

参考|vimdiff|可以了解关于diff模式更多的内容．

08.8 其它

'laststatus'选项用于指定何时最近使用的窗口会有一个状态行：

List

0	永远没有
1	只有分隔窗口时(默认值)
2	总是存在

很多要打开另一个文件的命令都有一个变体，可以新开一个窗口来打开指定的文件．对于命令行命令而言通常是在一般命令前面附加一个"s"． 例如":tag"可以跳转到一个tag，":stag"则是打开一个新窗口跳转到该tag．

对于Normal模式的命令是在命令前使用CTRL-W． CTRL-^可以跳转到前一个编辑的文件，CTRL-W CTRL-^则分隔出一个窗口来编辑前一个文件．

'splitbelow'选项用来控制新开的窗口出现在当前窗口的下面． 'splitright'则相应地使新开的垂直窗口出现在当前窗口的右边

分隔窗口命令还可以有一个位置修饰限定词作为前缀，用来指定新打开的窗口将出现的位置：

List

:leftabove {cmd}	当前窗口的左边或上面
:aboveleft {cmd}	同上
:rightbelow {cmd}	当前窗口的右边或下面
:belowright {cmd}	同上
:topleft {cmd}	当前窗口的上边或左边
:botright {cmd}	当前窗口的下面或右边

下一章： |usr_09.txt| 使用GUI

[1]译：获取差异之意． "diff get")可能更好些,不过它已经被别的命令用了("dg",键入这两个字符之后处于所谓pending状态，即等待键入命令的其余问题，如接下来键入的是3j则会删除自当前行开始连续4行

usr_09.txt Vim 6.3版.最后修改: 2001年09月03日

VIM用户手册--- 作者: Bram Moolenaar

使用GUI

Vim运行在普通终端上. GVim除了能做跟Vim一样的工作之外, 还有一些其它功能. GUI 提供了菜单, 工具栏, 滚动条等等. 本章的内容是关于GVim有别于Vim的GUI特性的.

下一章: |usr_10.txt| 大刀阔斧
前一章: |usr_08.txt| 分隔窗口
目 录: |usr_toc.txt|

09.1 GUI的各部分

你应该在桌面上有一个启动gVim的图标. 如果没有的话, 可以用下面两个命令之一来启动:

shell command
```
gvim file.txt
vim -g file.txt
```

如果还是不能启动那说明你的Vim没有GUI功能. 你要安装好先.

Vim会打开一个窗口并在其中显示"file.txt"的内容. 窗口的外观要看你的Vim版本而定. 基本上大致是下图的样子(用ASCII来画这个也只能这样了!).

Display

```
+----------------------------------------------------------+
| file.txt + (~/dir) - VIM                               X |   <- window title
+----------------------------------------------------------+
| File  Edit  Tools  Syntax  Buffers  Window  Help         |   <- menubar
+----------------------------------------------------------+
| aaa  bbb  ccc  ddd  eee  fff  ggg  hhh  iii  jjj         |   <- toolbar
| aaa  bbb  ccc  ddd  eee  fff  ggg  hhh  iii  jjj         |
+----------------------------------------------------------+
| file text                                              | ^ |
| ~                                                      | # |
| ~                                                      | # |  <- scrollbar
| ~                                                      | # |
| ~                                                      | # |
| ~                                                      | # |
|                                                    | V |
+----------------------------------------------------------+
```

最大的区域还是留给了文本显示． 这跟在终端中使用Vim一样． 只有部分文本的颜色或字体有些不同．

窗口标题

窗口顶部是它的标题．这由你的窗口系统负责显示． Vim会把当前文件的名字设为窗口的标题．首先是文件名，然后一些特殊字符和文件所在的目录． 其中特殊字符是这样的形式：

List

-	文件内容不可更改(例如一个帮助文件)
+	文件内容已被改变
=	文件是只读的
=+	文件是只读的，但内容已被更改

如果没有特殊字符的话就是一个未被更改的普通文件．

菜单条

你早就知道菜单是怎么回事，对不对? Vim中的菜单也一样，不过当然也有自己的特色．请先浏览一下菜单看看你会怎么使用． 以Edit/Global Settings菜单为例． 你会看到下面这样的菜单命令：

List

Toggle Toolbar	打开/关闭工具栏的显示
Toggle Bottom Scrollbar	打开/关闭底部滚动条的显示
Toggle Left Scrollbar	打开/关闭左边滚动条的显示
Toggle Right Scrollbar	打开/关闭右边滚动条的显示

在多数系统上你都可以把这些菜单剪切下来使它成为一个浮动窗口．选择顶级菜单，你会看到最一面的一行是一行点线．点击该行它会变成一个包含了该菜单下命令的浮动窗口． 该窗口在你关闭它之前都会一直存在．

工具栏

工具栏包含了最常用命令的图标．希望这些图标的选择都是可以望图知义的． 同时每个图标还有一个小提示描述它的功能(将鼠标移到图标上不要点击，稍停一下你会看到)

"Edit/Global Settings/Toggle Toolbar"菜单命令可以控制工具栏的存亡． 如果你从来都不用工具栏，可以在你的vimrc文件里这样设置：

ex command
```
:set guioptions-=T
```

该命令将移除'guioptions'选项中的"T"标志．通过该选项还可以控制GUI的其它组件是否显示． 请参考相关的帮助．

滚动条

默认情况下右边会有一个滚动条． 它的功能显而易见．当你分隔窗口时，每个窗口也都会有它自己的垂直滚动条． 你也可以通过"Edit/Global Settings/Toggle Bottom Scrollbar"菜单命令来打开位于底部的水平滚动条． 在diff模式下或者关闭了'wrap'选项(稍后会有详解)时水平滚动条就派上用场了．

如果有垂直分隔的窗口的话，只有最右边的窗口会有一个滚动条．而且你定位到左边的窗口时，滚动条所控制的还是最右边的窗口． 你可能要花些时间来习惯它[1]． 如果你经常用到垂直分隔的窗口． 可以考虑在窗口左边加一个滚动条． 这可以通过控制'guioptions'选项来实现：

ex command
```
:set guioptions+=l
```

该命令把"l"标志加到'guioptions'选项中去．

09.2 使用鼠标

标准化真是好东东．在微软Windows操作系统下，你可以用鼠标以通用的方式选取文本． X windows系统也有使用鼠标的标准．不幸的是，这

[1]译：在MS-Windows上，垂直分隔后窗口左边也会有一个滚动条，它控制除最右边窗口外所有窗口的滚动

两个标准本身并不相同. 幸运的是, 你还可以定制Vim. 你可以让以X windows的方式或微软的方式来使用你的鼠标. 下面的命令控制鼠标遵循X Window标准:

ex command
```
:behave xterm
```

下面的命令则使鼠标服从微软的标准:

ex command
```
:behave mswin
```

在UNIX系统上鼠标默认使用xterm标准. 对于微软Windows系统安装时可以进行选择. 关于这两套标准的更多内容, 请参考|:behave|. 下面是一个小结.

XTERM鼠标

List

左键单击	定位光标
左键拖动	在Visual模式下选取文本
中键单击	粘贴剪贴板的内容
右键单击	扩展被选择的文本到单击的位置

微软的鼠标行为

List

左键单击	定位光标
左键拖动	在Visual模式下选取文本
左键单击, 同时按下Shift键	扩展被选择的文本到单击的位置
中键单击	粘贴剪贴板的内容
右键单击	显示弹出菜单

鼠标的行为还可进一步调节. 如果你想进一步定制鼠标的话请参考下面的选项:

List

'mouse'	Vim在哪些模式中使用鼠标
'mousemodel'	控制鼠标单击的效果
'mousetime'	双击鼠标的间隔时间
'mousehide'	键入时隐藏鼠标
'selectmode'	控制如何可以进入Visual模式或选择模式

09.3 剪贴板

在|04.7|节中介绍了剪贴板的基本使用. 不过关于X-windows系统有还一个重要的不同: 它有两个地方供应用程序之间交换信息. MS-Windows没有此项功能.

在X-Windows系统中有一个"当前选择区". 这是当前被高亮显示的文本. 在Vim中这就是Visual区域(假设你用的是默认的设置). 你可以在另外的程序中直接粘贴这部分内容.

例如, 在本文中用鼠标选取几个单词. Vim会切换到Visual模式并高亮这被选取的文本. 现在启动另一个gVim, 不要带文件名参数, 这样它启动后会显示一个空的窗口. 点击鼠标右键. 前面被选取的部分将会被插入.

"当前选择区"会一直保持有效, 直到你下次又选取了另外的部分. 上例中, 在另一gVim中粘贴之后, 现在在该窗口选取一些内容, 你会注意到此前在第一个窗口中被选取的部分有所改变. 这意味着它不再是"当前选择区"了.

使用鼠标进行选取并不是必需的. 你也可以通过键盘上的命令进行Visual模式下的选取.

真正的剪贴板

现在轮到说另一个可以交换信息的场所了. 为避免混淆, 我们把它叫做"真正的剪贴板". 通常情况下"当前选择区"和"真正的剪贴板"都叫剪贴板, 你应该习惯这种叫法.

要把文本放到真正的剪贴板上, 还是先进行选取. 然后用Edit/Copy菜单命令. 现在文本就被复制到了"真正的剪贴板"上. 当然你看不到, 除非你有一个程序可以显示剪贴板的内容(比如KDE的klipper程序).

现在换到另一个gVim中, 将鼠标定位在某处后使用Edit/Paste菜单命令. 你会看到来自"真正的剪贴板"的内容已被插入.

二者并用

同时使用"当前选择区"和"真正的剪贴板"听起来就很混乱. 但实际上很有用. 我们用一个例子来说明. 使用其中一个gVim执行下面的动作:

1. 在Visual模式下选取两个单词.

2. 使用Edit/Copy菜单命令将它复制到剪贴板.

3. 在Visual模式下选取另一个单词.

4. 使用Edit/Paste菜单命令. 实际发生的是被选择的单词被前面的两个单词所取代
5. 将鼠标移至别处单击中键. 你会看到刚才覆盖了剪贴板内容的单个单词被插入.

如果你小心使用"当前选择区"和"真正的剪贴板"的话, 它们可以为你做很多事.

使用键盘

如果你不喜欢使用鼠标, 你可以用两个寄存器来访问"当前选择区"和"真正的剪贴板". "*寄存器指代"当前选择区".

要使被选取的文本变成"当前选择区", 使用Visual模式. 例如, 要选取整行的话按"V".

在当前光标前插入"当前选择区"的内容:

normal mode command
```
"*P
```

注意大写的"P". 小写的"p"是把文本放到当前光标的后面.

"+寄存器指代"真正的剪贴板". 例如, 要把自当前光标至行尾的内容放到该剪贴板:

normal mode command
```
"+y$
```

记住, "y"指yank, 这是Vim中对复制命令的叫法.

把"真正剪贴板"的内容放到当前光标之前:

normal mode command
```
"+P
```

跟"当前选择区"一样, 只不过+号代替了星号*.

09.4 选择模式

现在用在MS-Windows上的东西比X-Windows上的更多一些. 但两者都能做同样的事. 你已经知道了Visual模式. 选择模式类似于Visual模式, 因为它也用于选择文本. 但是两者有一个明显的区别: 键入文本时, 被选择的文本会被删除, 新键入的内容会取代它.

要用Select模式, 你得先打开它(对MS-Windows很可能已经是打开的了, 但你还是可以手工打开它):

ex command
```
:set selectmode+=mouse
```

现在用鼠标去选取一些文本. 它会象Visual模式下一样被高亮. 按下一个字符. 被选择的文本将被删除, 按下的字符取代了它. 现在你置身于Insert模式, 可以继续键入了.

因为键入任何文本都会抹去被选取区域的内容, 你不能用Normal模式下的"hjkl", "w"等等这些命令. 不过你可以用Shift功能键. <S-Left>(按住Shift键的同时按左光标键)将把光标左移. 被选取部分的变化跟在Visual模式下一样. 其它的Shift+光标键也行为正常. 还可以使用<S-End>和<S-Home>.

可以通过'selectmode'选项来调整Select模式的工作方式.

下一章: |usr_10.txt| 大刀阔斧

usr_10.txt Vim 6.3版.最后修改: 2004年03月12日

VIM用户手册--- 作者: Bram Moolenaar

大刀阔斧

第4章中介绍了几种对文件进行小幅改动的方法. 本章的内容是如何对文本作出大量改动的办法. Visual模式允许对被选的文本块进行多种操作. 也可使用一个外部程序来完成一些复杂的动作.

下一章: |usr_11.txt| 灾难恢复
前一章: |usr_09.txt| 使用GUI
目　录: |usr_toc.txt|

10.1 命令的记录与回放

"."命令可以重复最近一次的编辑动作. 但是如果你要做的操作远比这些小儿科复杂呢? 那就是Vim为什么要引入命令记录的原因. 使用命令记录分三个步骤[1]

1. 使用"q{register}"命令开始, 后续的动作将被记录进名为{register}的寄存器中. 给出的寄存器名字必需是a到z之间的一个字母[2]

2 执行你要执行的操作[3].

3. 按下q以结束对命令的记录(注意仅仅是q一个字符, 不要键入多余的字符).

现在你可以通过"@{recording}"命令来执行刚刚记录下来的记录宏了.

[1]译注: 很多貌似复杂的事情其实都只分三步, 如把大象装进冰箱:-)
[2]译注: 包括a和z
[3]译注: 象往常一样

下面的例子将演示如何实际运行该功能. 假如你有如下的文件名列表:

List
```
stdio.h
fcntl.h
unistd.h
stdlib.h
```

而你实际想要的结果如下:

List
```
#include "stdio.h"
#include "fcntl.h"
#include "unistd.h"
#include "stdlib.h"
```

将光标移动到第一行上. 接下来执行:

List
```
qa                      开始将后续的命令记入寄存器a
^                       将光标移动到行首
i\#include "<Esc>       在该行之前插入#include "
$                       移动到行尾
a"<Esc>                 在行尾加上"字符
j                       移到下一行
q                       停止记录
```

现在你已经将对第一行的操作完成了一遍, 对其它三行只需要执行3次"@"命令"@a"命令也可以加一个数字前缀[1], 这会使该记录被回放由该数字指定的次数. 在上面的例子中是:

normal mode command
```
3@a
```

移动并执行操作

也许实际情况是你在几个不同的地方要执行这些操作(而不象上例中是连续的4行). 这只需要你将光标定位到目标行, 然后再执行"@a"命令. 如果已经执行过"@a"命令, 下次重复执行只需再下"@@"即可. 这比"@a"更容易键入. 同样, 如果你上次执行的是"@b"那么"@@"命令也将重复"@b"的动作.

与"."方法相比, 记录回放有几个地方不同, 首先, "."命令只能重复一个动作. 而在上例中, "@a"重复的是好几个命令, 其次, "."命令只重

[1]译注: 就象绝大多数VIM命令一样

复最近一次改动的命令． 而执行一个记录宏允许你随时都可执行同样的操作．最后,你拥有多达26个寄存器可供使用． 也就是说，可以同时保存26个不同的命令宏．

使用寄存器

用于命令记录的寄存器与用于yank和删除命令的寄存器是同一个东西．所以还可以多种方式混合操作这些寄存器．

假设你在寄存器n中记录了一些命令． 执行时发现有些命令弄错了．当然你可以再重来一遍，但是还有其它办法补救：

List

G	到文件最后
o<Esc>	生成一个新行
"np	将寄存器n的内容置于该行．你会看到这些命令就象你键入的普通文本一样．
{edits}	修改错误的部分．这与编辑普通文本无异
0	到行首
"ny$	将正确的结果回存到寄存器n中
dd	删除这行草稿

现在你可以用"@n"来回放正确的命令宏了． (如果你的宏记录命令中包括有断行，调整上例中最后两个操作以确保将正确的结果存回寄存器n中)

向寄存器中追加内容

目前为止我们用到的还都是小写字母的寄存器． 要向寄存器追加内容而不是覆盖它，使用它的大写形式即可．

假设你已经记录了一个改变word的命令在寄存器c中． 它已经可以正常工作，但你又想让它搜索到下一个word继续编辑． 可以使用下面的命令：

normal mode command

```
qC/word<Enter>q
```

以"qC"命令开始，这使得被记录的命令将被追加到寄存器c中，而不是覆盖它当前的内容．

这种方式对命令记录和一般的yank，删除操作都有效． 比如你要把几行的内容收集到一个寄存器中去，以小写来yank第一行：

normal mode command

```
"aY
```

现在移动到第二行，执行:

normal mode command
```
"AY
```

对你要收集的行重复执行这个操作. 现在寄存器a就会包括所有这些行的内容.

10.2 替换 *find-replace*

":substitute"命令可以对一个指定范围的区域执行替换操作. 它的通用形式如下:

ex command
```
:[range]substitute/from/to/[flags]
```

该命令将由[range]指定的行中的字符串"from"替换为"to". 比如你要把所有行中的"Professor"替换为"Teacher":

ex command
```
:%substitute/Professor/Teacher/
```

> 备注: 一般人都不会把:substitute完整拼出来，使用它的缩略形式":s"就可以了. 本文中其余部分将都使用这种形式.

命令之前的"%"指定该命令将作用于所有行上. 不指定一个范围的话，":s"将只作用于当前行. 下一节|10.2|中将会讲述关于指定范围的详细内容.

默认情况下，":substitute"命令只会替换一行中第一次被发现的目标字符串. 比如，上一个命令将把

Display
```
Professor Smith criticized Professor Johnson today.
```

变为:

Display
```
Teacher Smith criticized Professor Johnson today.
```

要改变一行中所有符合的目标字符串，可以在命令后加"g"标志加以修饰.

ex command
```
:%s/Professor/Teacher/g
```

对此例来说结果将是:

Display
```
Teacher Smith criticized Teacher Johnson today.
```

其它可用于该命令的修饰标志还有p(列印)，使":substitute"命令列出每个被它改变的行. c(确认)标志告诉":substitute"命令要在执行每个替换前要求用户确认. 执行下面的命令:

ex command
```
:%s/Professor/Teacher/c
```

Vim会在找到第一个"Professor"后显示下面的信息并要求你的回答:

Display
```
replace with Teacher (y/n/a/q/l/^E/^Y)?
```

此时，你可以有几种答案[1]:

List

y	好吧，替换吧
n	不，这个先留着
a	别问了，全部换掉吧(这群教授都不够格?? :-))
q	退出，剩下的也不要管了
l	把现在这个改完就退出吧
CTRL-E	向上滚屏一行
CTRL-Y	向下滚屏一行

例子中的目标字符串"from"实际上可以是任何合法的正则表达式. 与搜索命令所用的正则表达式一样[2]. 比如，下面的命令只会把一行行首的"the"替换为"these":

ex command
```
:s/^the/these/
```

如果你要替换的字符串中包含了斜杠/，就需要在它前面加一个反斜杠，一个更优雅的办法是用另一个字符替换/作为命令中各部分的分隔符[3]:

[1]译注: 关于下面的CTRL-E CTRL-Y. 用户确认是否要进行替换往往需要查看上下文的信息，如果当前被找到的行是在屏幕最底部，则没办法看到其下的内容,所以有这一功能

[2]译注: 整个Vim中用的都是同一个正则表达式引擎，所以完全通用

[3]译注: 确保被选中的新分隔符不会出现在你的源/目标字符串中

ex command
```
:s+one/two+one or two+
```

10.3 命令作用范围

对于":substitute"命令和很多其它的":"命令，可以指使它们作用于一些行上，这叫命令的作用范围. 作用范围的最简单形式是两个以数字表示的行号. 如下:

ex command
```
:1,5s/this/that/g
```

该命令将对第1到第5行的文本执行替换操作. 也包括第1行和第5行. 这样的作用范围总是放在命令的最开始

单个的数字指示命令将只作用于由该数字指定的行上:

ex command
```
:54s/President/Fool/
```

有一个命令在你不指定作用范围时默认是对整个文件进行操作. 要使它们只作用于当前行上，可以在命令前放一个"."[1]. ":write"就是这种命令的典型. 不指定作用范围，它将写入整个缓冲区的内容. 下面的命令使它只把当前行写入指定文件:

ex command
```
:.write otherfile
```

第一行的行号一定是1. 但是最后一行呢? "$"用于代表最后一行. 比如，下面命令替换当前行到最后一行中所有的yes为no:

ex command
```
:.,$s/yes/no/
```

所以，前面用到的"%"范围指定符号，实际上等价于"1,$".

使用一个搜索模式来指定作用范围

假设你正在编辑一本书中的某一章内容，你想替换本章中所有的"grey"为"gray". 但又不殃及其它章里的"grey". 你知道只有在章与章的边界才会有"Chapter"这个词出现在一行的行首，所以命令是:

[1]译注: 代表当前行

ex command
```
:?^Chapter?,/^Chapter/s=grey=gray=g
```

你看，这里用了两个搜索模式. 第一个"?^Chapter?" 向后查找，"/^Chapter/"向前. 为了避免眼花liao乱的斜杠/，":s"命令使用"="字符作为分隔符.

增与减

其实上例中还略有瑕疵: 如果下一章的标题中刚好含有"grey"那么它也将会被替换掉. 也许你要的就是这种效果，但如果不是呢? 可以对命令的作用范围指定一个偏移作为微调: 找到一个符合的模式并使用它上面的一行:

ex command
```
/Chapter/-1
```

可以用任何数字来替换1. 要定位匹配模式其下的第2行:

ex command
```
/Chapter/+2
```

作用范围的上下偏移也可用于以其它形式:

ex command
```
:.+3,$-5
```

这个范围从当前行其下的第3行开始，到倒数第6[1].

使用标记

使用标志可以免于上面的行号计算: 在某处作上标记，然后以此标记来指定作用范围.

用第3章里的办法作上标记. 如用"mt"来标记一个范围的开始，"mb"标记它的结束. 然后就可以这样指定这个范围:(包括标记本身所在的行)

ex command
```
:'t,'b
```

Visual模式与范围

如果你在Visual模式下选定了文本后按下了":"，你将会看到如下命令:

[1]译注: $-1是倒数第一行，编辑器里也有臭名昭著的'offset by one'错误

ex command
```
:'<,'>
```

现在你只需直接键入命令，作用范围已由'<,'>指定好了，它代表你在Visual模式下选定的文本所在的范围.

> 备注：当你用Visual模式或用CTRL-V去选择文本块时，该命令指定的作用范围仍是以行为基本单位. 这一点会在Vim的后来的版本中改进.

'<和'>实际上就是标记，分别代表一个Visual选择区域的开始和结束. 退出Visual模式后这两个标记仍然保持，直到下一次进入Visual模式. 所以你还可用"'<"命令来跳转到你一次在Visual模式时选定的文本区域的开始处. 也可以混合使用多种方法指定作用范围：

ex command
```
:'>,$
```

它表示自上一次Visual模式时选定的文本区域的开始处到文件尾这样一个区域.

以数字指定行数

如果你已经知道要使命令使用于几行内容，可以直接按下这个数字，然后按":". 比如，按下"5:"，你会看到：

ex command
```
:.,.+4
```

你要做是直接键入命令. 它将作用的范围是"."(当前行)到".+4"(自当前行到向下4行). 所以一共是5行.

10.4 全局命令

":global"命令是Vim最强大的功能之一. 它允许你找到符合某个匹配模式的行然后将命令作用其上. 下面是其一般形式：

ex command
```
:[range]global/{pattern}/{command}
```

乍一看它与":substitute"命令很像. 但是，这里执行的是由{command}指定的命令

备注: ":global"中所谓的命令都必需是以":"开始的命令行命令, Normal模式下的命令不能直接使用. |:normal|命令可以间接地让你使用Normal模式下的命令.

假设你想把C++风格的注释中的所有"foobar"替换为"barfoo"(这些注释将以"//"开始):

ex command
```
:g+//+s/foobar/barfoo/g
```

该命令以":g"开始, 它是":global"的缩写, 就象":s"是":substitute"的缩写一样. 接下来是以加号分隔的搜索模式. 因为我们要搜索的内容中包括有斜杠/, 所以此处用加号来分隔命令的不同部分. 最后是将"foobar"替换为"barfoo"的命令.

全局命令的默认作用范围是整个文件. 所以此例中没有指定作用范围. 这一点与命令":substitute"不同, 它在没有指定作用范围时默认对当前行一行起作用.

上面给出的命令还不足以精确达到它的目标, 因为它也匹配到那些"//"出现在一行中间的情形, 这时如果在"//"之前也出现了"foobar", 那么它也会被误换掉[1].

与":substitute"中对正则表达式的应用一样, 全局命令中也可使用任何Vim中合法的正则表达式. 接下来你会学到更多复杂的正则表达式技巧.

10.5 Visual block 模式

使用CTRL-V可以进入一种特殊的选择模式, 在此模式下你可以选择一个矩形的文本块. Vim提供了一些特别的命令来操纵这个文本块[2].

在Visual block模式下"$"命令会让每一行的被选择区域扩展到该行的末尾, 不管这些行的长短是否参差不齐. 这种选择状态持续到你发出下一个改变水平选择域的命令. 所以使用命令"j"会保持这种状态, 而"h"命令则会停止它.

插入文本

命令"Istring<Esc>"会在每行中插入相同的文本, 插入位置在被选择块的左边. 具体过程是以CTRL-V进入Visual block模式. 然后移动光标

[1]译注: 如下:
puts("foobar"); // this line contains a foobar
如何解决这个问题留给读者
[2]译注: 下面解释的"$"命令将使被选择区域看起来并不是一个矩形, 这是一个特例

来调整被选择的区域. 接下来键入I命令进入插入模式, 键入你要插入的文本. 在你键入文本的过程中, 被键入的内容只会同时显示在文本块的第一行中. 一旦你按下<Esc>来结束插入, 刚刚键入的内容就会奇迹般地出现在被选择文本块的每一行中. 如:

Display
```
include one
include two
include three
include four
```

将光标移到"one"中的"o"上, 然后按下CTRL-V. 用"3j"命令将选择区域扩大到向下3行, 到单词"four"上. 现在你选择了一个纵跨4行的文本块. 开始键入:

normal mode command
```
Imain.<Esc>
```

结果将是:

Display
```
include main.one
include main.two
include main.three
include main.four
```

如果选择的文本块的跨度包含一些太短的行以致于它的内容不能出现在文本块中, 那么被插入的文本会跳过这些行. 如下例中, 选择一个同时包含第一行中"long"和最后一行中的"long"的文本块, 这个文本块就没有包含第2行的任何内容:

Display
```
This is a long line
short
Any other long line

          ^^^^ selected block
```

现在键入命令"Ivery <Esc>". 结果是:

Display
```
This is a very long line
short
Any other very long line
```

你看, 含有"short"的第二行没有被插入任何东西.

如果你插入的过程中进行了换行，那么"I"命令将会象Normal模式下一样，只影响文本块的第一行.

"A"命令类似于"I"，只不过它是在文本块的最右边追加文本． 对"A"命令还有一种特殊情况： 选择一个文本块然后按下"$"使文本块扩展到每行的末尾. 然后用"A"命令追加一些文本． 同样以上例说事，按下命令"$A XXX<Esc>"，结果如下：

Display
```
This is a long line XXX
short XXX
Any other long line XXX
```

要收此宏效必需要用"$"命令，Vim会记住你到底有没有用它． 如果你只是把光标移到最后同时是最长的行的那一行末尾[1]，可别想达到这样的效果.

改变文本

Visual block模式下的"c"命令会删除被选择的文本块，然后你会身处Insert模式，键入改变后的内容. 键入的内容会被插入在文本块的第一行上[2] 还拿上面的例子来说，假设你选择了包含第一第三行"long"单词的文本区域，然后键入"c_LONG_ <Esc>"命令，你会看到：

Display
```
This is a _LONG_ line
short
Any other _LONG_ line
```

与"I"命令对中间的短行的效果一样： 它不受影响，同时，新键入的文本中也不能有换行.

"C"命令会删除文本块最左边至每一行末尾的所有内容[3] ，然后你又会在Insert模式下，键入的文本会追加到每一行的最后[4]． 同上例，键入命令"Cnew text <Esc>"，结果如下：

Display
```
This is a new text
short
Any other new text
```

[1]译注： 看起来可能会与"$"相同，如下：
```
This is a long line XXX
short XXX
Any other long line XXX
```

[2]译注： 当然又是在<Esc>之后才能看到效果
[3]译注： 注意不是文本块的末尾
[4]译注： 实际上，中间的短行仍被排除在外

注意，虽然说文本块中只包含了单词"long"，文本块后面的内容还是全部被删除了． 所以此时只有文本块的左边界对这个命令有影响．同样地，中间的短行被排除在外．

Visual block模式下还有其它一些改变文本内容的命令：

List

```
~      交换大小写     (a -> A, A -> a)
U      将小写变大写   (a -> A, A -> A)
u      将大写变小写   (a -> a, A -> a)
```

以一个字符填充

"r"将使整个文本块的内容全部以一个字符来填充． 同上例，选择了文本块后按下"rx"命令：

Display

```
This is a xxxx line
short
Any other xxxx line
```

备注： 如果你要选择一个延伸到行尾后面的文本块，请查看第25章'virtualedit'部分的内容．

左右移动

">"命令会使你的被文本块向右移动一个"shift 单位"[1]，空出来的部分放置以空格．被移动的区域始自文本块左边界． 同上例，">"命令后结果如下：

Display

```
This is a          long line
short
Any other          long line
```

移动的多寡由选项'shiftwidth'指定． 欲使之移动4个空格：

ex command

```
:set shiftwidth=4
```

[1]译注： 一个"shift 单位"由:set sw=N 指令指定，N是一个自然数，请参考:h 'sw'

"<"命令使文本块向左移动一个"shift 单位". 不过它不象">"一样姿意，它受限于文本块左边的空间，所以如果文本块左边的空白区域短于一个"shift 单位"，它也无能为力[1].

将多行内容粘接起来

"J"命令使文本块纵跨的所有文本行被连接为一行. 换行符不存在了，实际上，换行符,以及每行的前导空白和后缀空白[2]都将被替换为单个的空格. 连接后的行尾将被放两个空格[3].

还用那个我们熟知的例子吧. 现在"J"命令之后结果将是

Display
```
This is a long line short Any other long line
```

"J"命令并不要求你一定在Visual block模式下作块选择. 你用"v"命令或"V"命令进行选择时效果完全一样[4]

如果你想保留那些前导空白和后缀空白，用"gJ"命令来代替"J"

10.6 读写部分文件

在你写e-mail时，也许会想引用来自另一个文件中的内容. 这里可用":read {filename}" 命令. 这样被读入文件的内容就被放在当前行的后面. 以下面的几行内容为例吧:

Display
```
Hi John,
Here is the diff that fixes the bug:
Bye, Pierre.
```

现在把光标移到第二行上，然后键命令:

ex command
```
:read patch
```

名为"patch"的文件的内容将被插入到这里，结果可能是:

[1]译注: 此时命令将没有任何效果，只要文本块中有一行属于这种情况，则整个文本块都不会左移，尽管有些行有足够的空间来左移

[2]译注: 此处用前导空白指出现在一行最前面的一个或多个连续的空格或<Tab>，后辍空白指一行最后的一个或多个连续的空格或<Tab>用正则表达式来描述，分别是/^\s\+/和/s+$/, TODO: 编辑器描述语言

[3]译注: ??

[4]译注: 因为它关心的文本单位是行

Display
```
Hi John,
Here is the diff that fixes the bug:
2c2
<       for (i = 0; i <= length; ++i)
---
>       for (i = 0; i < length; ++i)
Bye, Pierre.
```

":read"命令还可接受一个行范围. 被读入的文件被放在这个范围的最后一行上. 所以":$r patch"将会把文件"patch"的内容追加到当前文件的最后. 如果你要把文件放在第一行的上面呢? 答案是特殊行号0. 当前行号为0的行并不存在. 如果你以此行号作为目标行执行大多数其它命令, 你会得到一个错误信息. 但对"read" 命令而言允许现在这样的命令:

ex command
```
:0read patch
```

文件"patch"将会被放在第一行的上面.

写入指定范围行

要写入指定范围行, 可以用命令":write". 没有指定一个范围时该命令将写入整个文件的内容. 指定一个范围的话它就只写入指定的行:

ex command
```
:.,$write tempo
```

这个命令将从当前行到文件尾的内容写入文件"tempo", 如果文件"tempo"已经存在, 你会收到一条错误信息, Vim会保护你意外地覆盖掉其它文件. 如果你要的就是覆盖它, 可以在"write"命令后放一个!

ex command
```
:.,$write! tempo
```

注意: !必需紧挨着放在":write"之后, 中间不要有任何的空白. 否则它会被解释为一个过滤命令. 下一章会解释什么是过滤命令[1].

向目标文件里追加

本章的第一小节中讲到如何将多行的内容收集到一个寄存器中去, 同样也可以将这个办法用于将它收集到一个文件中去. 写入第一行的内容:

[1]译注: 过滤命令

ex command
```
:.write collection
```

现在把光标移到第二行上去，键入命令：

ex command
```
:.write >>collection
```

">>"告诉Vim不要把"collection"看作一个新文件往里写东西，而是把要写的东西追加在它的后面. 当然，你可以随意地多次执行这个命令.

10.7 格式化文本

要是你在键入文字的时候每行的内容能自动调节到适应当前窗口的大小该有多好. 'textwidth'选项即用于实现这一功能[1]：

ex command
```
:set textwidth=72
```

还记得我们在vimrc文件的例子中为每个文件指定了该选项吗? 如果你用到了那个vimrc，你就已经在使用这一选项了.

ex command
```
:set textwidth
```

现在每一行都会自动调整到只包含最多72个字符，但是如果你是在一行的中间删除或插入内容，Vim可管不了这么多. 这样你的行还是可能变得太长或太短.

下面的命令告诉Vim格式化你的当前文本段：

normal mode command
```
gqap
```

该命令以"gq"开始，这是Vim中的一个操作符. 接下来是"ap",它代表"a paragraph"这样一个文本对象，文本段与文本段之间的分隔标志是一个空行.

> 备注： 包含了空白字符的行可不是这里说的空白行，这经常被很多人忽略!

[1]译注： 中文用户怎么办??

除了"ap"你还可以用移动命令或其它指定文本对象的办法. 如果你的整个文件都已经被正确地分为各个文本段，可以用下面这个命令来格式化整个文件:

normal mode command
```
gggqG
```

"gg"会首先定位到第一行，然后"gq"告诉Vim要格式文本了，"G"移动操作符跳转到最后一行，连起来的意思就是格式化整个文件.

如果你并没有把你的文本内容分隔为一个一个的文本段，还可以只格式化一部分文本行. 将光标置于要格式化的第一行上，命令"gqj"会格式化当前行和它下面的一行. 如果第一行太短，下一行的内容就会追加在它后面. 如果它太长，长出来的单词[1]会被放到下一行去. 接下来你可以使用"."命令来重复刚才的操作，直到你格式化到满意为止.

10.8 改变大小写[2]

如果你已有的文本中所有的section header[3]都是小写. 你想把"section"全部变为大写. "gU"命令可担此任. 当光标置到第一列上[4]

"gu"命令反"gU"之道而行之:

Display
```
                         guw
    SECTION header      ---->      section header
```

还可以用"g~"来使所有字母的大小写反个过，大写变小写，小写变大写. 因为它们是操作符命令，所以可以搭配使用所有的移动命令，或者在Visual模式下先选择文本对象然后执行该操作[5]. 要使一个操作子命令作用于以行为单位的对象你可以键入该操作子两次. 比如，删除操作子是"d"，所能删除一行的命令是"dd". 同样，"gugu"使一整行变为小写. 此外，它还可简写为"guu". "gUgU"简写为"gUU"，"g~g~"简写为"g~ "[6]. 如:

Display
```
                              g~~
    Some GIRLS have Fun      ---->   sOME girls HAVE fUN
```

[1]译注: Vim格式化时不会把你的单词从中间打断，它格式化时不会在一个单词中间断行，用它的正则表达式描述，就是/\w\+/

[2]译注: 中文没有这个麻烦

[3]译注: 节标题

[4]译注: 真正意思是把光标放在你要使之变为大写的单词section的第一个字母上

[5]译注: Vim命令还细分为不同的种类，operator command可以搭配以motion 命令，和指定text object?? 暂译为"操作子命令"，因为数学中的操作子通常要搭配一个操作数，比如a + b，所以操作子命令可理解为需要搭配一个作用对象的命令

[6]译注: 这里没有直译，因为操作子命令还可另外搭配一个数字作为命令的重复次数，所以"3dd"会删除连同当前行在内的3行

10.9 使用外部程序

虽然Vim有一个几乎无所不能的强大的[1]命令集. 但是还是有一些任务如果用一个外部命令来做会更好或更快一些.

命令"!{motion}{program}"以一块文本为对象将它们通过管道送至一个外部程序. 换句话说, 由{program}指定的外部程序, 接受由{motion}命令指定的文本块作为输入, 以它的输出来替换{motion}指定的文本块.

如果你对UNIX的过滤程序不熟的话, 这样的解释还是令人费解, 还是看一个例子吧. sort命令将一个文件排序. 执行下面的命令会使未经排序的文件input.txt被排序后写入文件output.txt. (这个例子同时适用于UNIX和Microsoft Windows).

shell command
```
sort < input.txt > output.txt
```

现在回到Vim, 看看如何在这里做同样的事. 假设你要将第1-5行的内容排序, 首先将光标置于第1行上. 然后键入如下命令:

normal mode command
```
!5G
```

"!"告诉Vim你要执行一个过滤操作了. Vim希望接下来收到你继续键入的移动命令, 以此决定你要将哪个区域的文本块送至过滤程序. "5G"命令告诉Vim到第5行去, 所以Vim可以据此判断你要过滤的内容是第1行(也是当前行)到第5行.

由于这是一个过滤操作, 所以此时Vim会把光标放到命令行模式[2]. 接下来你可以键入过滤器的名字, 这里是"sort". 所以, 整个命令如下:

normal mode command
```
!5Gsort<Enter>
```

结果是sort程序将前5行排序. 将输出替换这5行的内容.

Display
```
line 55                 line 11
line 33                 line 22
line 11       -->       line 33
line 22                 line 44
line 44                 line 55
last line               last line
```

[1]译注: 同时也是庞大的

[2]译注: {motion}命令完之后Vim自动从Normal模式转入命令行模式, 这样也更方便键入整个命令中其余的部分,这个部分可能会很复杂, 在Normal模式下键入很容易出错, 还有其它一些类似的操作会自动从Normal模式转到命令行模式

"!!"命令过滤当前行的内容. 在Unix系统中"date"命令会显示当前的日期和日间. "!!date<Enter>"会以"date"命令的输出来替换当前行的内容. 要向文件里加入一个时间戳时这一命令很有用.

出了问题呢?

在Vim中执行如上的过滤操作需要知道一些shell的有关情况. 如果在使用过滤程序时遇到问题, 可以考虑检查下面一些选项的设置:

'shell'	指定Vim用于运行过滤程序的shell
'shellcmdflag'	该shell的参数
'shellquote'	用于分隔shell与过滤程序时成对包围起过滤程序的字符
'shellxquote'	用于分隔shell与过滤程序和重定向符号时成对包围起过滤程序和重定向符号的字符
'shelltype'	shell的类型(只对Amiga有用)[1]
'shellslash'	在命令中使用反斜杠(只对MS-Windows这样的系统有用)
'shellredir'	用于将命令输出重定向到文件的字符串

在Unix上使用过滤程序很少会碰到问题, 因为有两种shell: "sh"派和"csh"派. Vim会检查'shell'选项看是否包含了"csh"并自动设置相关的选项,

但在MS-Windows上, 有很多不同的shell, 所以你要手工调整这些选项来让过滤功能正常动作. 请查看上面这些相关选项的帮助以了解更多信息.

读取命令的输出

如下命令可以读取当前目录下的内容: Unix:

ex command
```
:read !ls
```

MS-Windows:

ex command
```
:read !dir
```

"ls"或"dir"命令的输出被Vim捕获并插入到当前当标下面. 这与读取一个文件很类似, 只有特殊的"!"用以告诉Vim接下来的是一个命令.

过滤程序也可以有自己的参数. 也可以指定一个范围告诉Vim把命令到的输出放到哪里:

ex command
```
:0read !date -u
```

这个命令会在文件开头插入UTC格式的当前日期时间(如果你的date命令支持"-u"参数的话)注意这与"!!date"命令的不同，"!!date"是替换当前行的内容，而":read !date"则是将输出结果插入到文件中.

将文本写入一个命令

Unix命令"wc"会统计字符数，单词数，行数. 要统计当前文件中的单词数，使用命令:

ex command
```
:write !wc
```

这与前面出现的"write"命令一样，但是不是写入一个文件名，而是一个"!"和一个外部程序名. 被写入的内容会通过标准输入送入指定的命令中. 输出结果大致是这样[1]:

Display
```
4      47     249
```

"wc"命令没有过多的描述这3个数字的意义. 这是说当前文件有4行，47个单词，共249个字符.

看一下下面敲错这个命令时会怎样:

ex command
```
:write! wc
```

这将会强制覆盖当前目录下的"wc"[2]. 这里的空白至关重要!

重绘屏幕

如果外部命令运行时发生了错误，整个屏幕输出可能会被弄乱. Vim自己在处理屏幕上何处何时需要重绘方面非常有效，但它还是没办法知道别的程序会做些什么. 下面的命令可以告诉Vim现在重绘一下屏幕显示:

normal mode command
```
CTRL-L
```

下一章: |usr_11.txt| 灾难恢复
版 权: 请参考|manual-copyright| vim:tw=78:ts=8:ft=help:norl:

[1]译注:根据文本内容的不同而不同
[2]译注: 如果它存在并且权限允许的话

usr_11.txt Vim 6.3版.最后修改：2004年04月23日

VIM用户手册--- 作者： Bram Moolenaar

灾难恢复

你的电脑死过机吗? 就在你辛辛苦苦编辑了几小时后? 别急! Vim已在磁盘上保留了充足的信息来恢复你的大部分工作. 本章将讲解Vim是如何利用交换文件来恢复你的劳动成果的.

下一章： |usr_12.txt| 奇技淫巧
前一章： |usr_10.txt| 大刀阔斧
目　录： |usr_toc.txt|

11.1 基本方法

多数情况下恢复文件是简单的，如果你知道要恢复的文件名(当然也要保证你的硬盘还会转)，就可以在启动Vim时指定一个"-r"参数:

shell command
```
vim -r help.txt
```

Vim会读取交换文件(这正是存放你已编辑的文件的地方)以及你的原文件的一些信息. 一切正常的话，你会看到下面的信息(当然文件名会不同):

Display
```
Using swap file ".help.txt.swp"
Original file "~/vim/runtime/doc/help.txt"
Recovery completed. You should check if everything is OK.
(You might want to write out this file under another name
and run diff with the original file to check for changes)
Delete the .swp file afterwards.
```

为安全起见，最好把它先存成另一个文件:

ex command

```
:write help.txt.recovered
```

你可以比较一下两个文件看看上次结束的地方是不是你想要的地方. Vimdiff程序用于比较两个文件的不同是再好不过了|08.71|. 查看原文件可以了解最近的一个版本是什么样子(也就是电脑死机前你保存过的版本). 看一下有没有丢失一些内容(有可能哪出了问题连Vim 也恢复不了)

如果Vim在恢复时给出了警告信息, 那可要特别注意这些警告, 尽管这种情况很少发生.

通常最后的少量改动无法恢复. Vim会在你连续4秒不键入内容时跟磁盘同步一次, 或者是连续键入了200个字符之后. 这可以通过'updatetime'和'updatecount'两个选项来控制. 所以如果系统有所改动但Vim还没有同步时发生了宕机, 那这一部分内容就无法恢复了.

如果你在编辑一个没有指定文件名的缓冲区时死机了, 可以通过一个空字符串作为要恢复的"文件名":

shell command

```
vim -r ""
```

确保你所在的目录是正确的, 否则Vim会找不到正确的交换文件.

11.2 交换文件在哪?

Vim可以在几个地方存放交换文件. 通常它跟原文件同一个目录. 要找到交换文件, 可以先切换到某个目录然后用下面的命令:

shell command

```
vim -r
```

Vim会列出所有找到交换文件. 它也会查看其它用来存放交换文件的目录来找到当前目录下的文件的交换文件[1]. 除此之外的其它目录就不会被搜索了, 同时也不会遍历当前的目录树[2].

屏幕输出大致象这样:

[1]译: 比如当前目录是~foo/, 其下有一个文件叫readme.txt, 虽然该目录下没有名.readme.txt.swp的交换文件, 但Vim还可能去/tmp 目录下看有没有可能在此存有readme.txt的交换文件

[2]译: 指搜索其子目录

Display
```
Swap files found:
   In current directory:
1.    .main.c.swp
          owned by: mool   dated: Tue May 29 21:00:25 2001
         file name: ~mool/vim/vim6/src/main.c
          modified: YES
         user name: mool   host name: masaka.moolenaar.net
        process ID: 12525
   In directory ~/tmp:
      -- none --
   In directory /var/tmp:
      -- none --
   In directory /tmp:
      -- none --
```

如果有好几个交换文件看起来都差不多，Vim会以全部列出这些交换文件，请你从中选择一个来进行恢复．此时要小心这些交换文件的日期信息．

如果你实在难以决断到底用哪个交换文件，那就一个一个试看看哪一个恢复后的内容正是你想要的．

指定交换文件

如果你能确切知道要用的交换文件名，你也可以在恢复时明确指定该文件．Vim会根据交换文件名找出原文件名．

例如：

shell command
```
vim -r .help.txt.swo
```

这同样适用于交换文件位于另一个非常规的目录中的情况． 如果这还不行，那就根据Vim报告的文件名把它改名． 查看'directory'可以获知Vim在哪些目录下存放交换文件．

> 备注：Vim会在'dir'选项指定的目录中搜索相应的"filename.sw?"文件．如果通配符不能正常工作(比如'shell'选项设置不当时)，Vim还是会试着查找名为"filename.swp"的文件．这样还是找不到的话，你就要自己指定要用的交换文件了．

11.3 是不是死机了？

Vim会尽量防止你做错事,. 假设无辜的你正想编辑一个文件, 希望Vim象往常一样显示该文件的内容. 但它却给出了一大堆这样的东西:

Display
```
        E325: ATTENTION
Found a swap file by the name ".main.c.swp"
          owned by: mool   dated: Tue May 29 21:09:28 2001
         file name: ~mool/vim/vim6/src/main.c
          modified: no
         user name: mool   host name: masaka.moolenaar.net
        process ID: 12559 (still running)
While opening file "main.c"
             dated: Tue May 29 19:46:12 2001

(1) Another program may be editing the same file.
    If this is the case, be careful not to end up with two
    different instances of the same file when making changes.
    Quit, or continue with caution.

(2) An edit session for this file crashed.
    If this is the case, use ":recover" or "vim -r main.c"
    to recover the changes (see ":help recovery").
    If you did this already, delete the swap file ".main.c.swp"
    to avoid this message.
```

这是因为编辑文件之前Vim会检查是否存在该文件的交换文件. 如果有, 那一定是出了状况. 可能是下面的两种情况:

1. 另一个Vim会话正在编辑该文件. 看看上面给出的信息中带有"process ID"的那一行. 它可能是这样:

Display
```
process ID: 12559 (still running)
```

"(still running)"表明同一台电脑上另一进程正在编辑此文件. 在非Unix系统上你可不会看到这么多的提示. 如果另一会话是通过网络编辑该文件, 那你也不会看到这段提示. 因为进程是运行在另一台电脑上. 这两种情况下你要自己想办法找出原因.

如果另一个Vim也在编辑该文件, 你再不顾一切地编辑它的话, 被编辑的文件就会有两个版本. 最后保存的版本将覆盖前一个的内容, 总会有人痛失一切. 所以最好还是礼让三先, 退出Vim.

2. 交换文件可能是肇因于上一次的系统崩溃或Vim自己崩溃. 详察一下Vim给出的日期信息. 如果交换文件比你要编辑的文件更新, 而且给出了下面的消息:

Display
```
modified: YES
```

此时很可能是某个Vim会话崩溃了，这还有得救.

如果交换文件的日期比要编辑的文件旧，那说明要么该文件在崩溃之后又被更改过.(比如你此前已经恢复过，但还留着交换文件)，要么文件的最后存盘时间是崩溃之前，但却是在交换文件最后存盘时间之后(算幸运了，老的交换文件也不再需要了). 这时Vim会警告你说:

Display
```
NEWER than swap file!
```

不可读的交换文件

有时候交换文件下面会出现这样的字样:

Display
```
[cannot be read] ~
```

这是好是坏? 要视情况而定.

如果上一次编辑过程中没有对文件做出任何改变，那是好事. 这时交换文件的明细情况会显示其长度为0. 你可以简单地删除它.

如果是因为你没有该交换文件的读权限就有点不妙. 你可以以只读方式浏览文件[1]，或者退出. 在多用户系统上，如果你对文件的最后一次修改时是以另一个账号登入系统的，退出系统后再以该账号登入就可以去除"read error". 或者你应该看看是谁最后一次编辑过这个文件(或者还正在编辑)并跟他谈一谈.

如果是因为磁盘的物理错误引起的那就非常糟糕了. 幸运的是这种情况很少发生. 你可以先试着以只读方式打开文件(如果可以的话)，看看对文件的修改有多少被丢失了. 如果正是你要对该文件负责，就准备好重来吧.

怎么办?

如果你的VIM支持对话框的话，它会以下面的形式提示你作出选择:

Display
```
Swap file ".main.c.swp" already exists!
[O]pen Read-Only, (E)dit anyway, (R)ecover, (Q)uit, (A)bort, (D)elete it:
```

O 以只读方式打开.

如果你只想查看文件内容不想恢复的话就选这个. 或者你知道别人正在编辑该文件，你不过想看一下它的内容而已.

[1]译: 没有读权限怎么读?

E 还是要编辑.

小心! 如果该文件正被另一Vim编辑, 你很可能会得到两个版本. 此时Vim会警告你, 但最好还是安全第一, 以和为贵.

R 从交换文件中恢复.

如果你确信交换文件中的内容正是你要找回的东西就那用这个.

Q 退出.

这样就免于编辑该文件了. 如果有另一个Vim会话正在编辑最好是选择退让. 如果你是刚刚启动Vim, 这个选择会同时退出Vim. 如果启动时打开了好几个窗口, Vim只有在第一个窗口遇到这种情况时才退出[1]. 如果是在使用一个编辑命令时选择退出, 该文件就不会被继续载入, 系统回到此前的编辑状态.

A 丢弃.

类似于退出, 但它同时会撤消对后续命令的执行, 这在载入一个脚本编辑多个文件时比较有用. 如打开一个多窗口的编辑会话时.

D 删除交换文件.

只有确信你已不再需要这个交换文件时才应做此选择. 比如, 交换文件里没有包含任何新的改动, 或原文件比交换文件还新. 在Unix上只有创建新交换文件的进程不再运行时才会给出这一选择.

如果你没看到对话框(你运行的是不支持对话框的Vim), 就要手工恢复了. 下面的命令执行恢复:

ex command
```
:recover
```

Vim并不总是能正确地检测到交换文件的存在. 比如另一编辑该文件的会话将交换文件放入了另一目录或者不同机器对被编辑文件的路径的理解不同. 所以不要什么都指望Vim.

如果你实在是不想看到这样的警告信息, 你可以在'shortmess'选项中加入"A"标志[2]. 但是多数情况下这一警告还是十分有用, 所以建议保留.

11.4 进一步的学习

[1]译: 否则Vim会继续打开其它文件进行编辑
[2]译: A代表ATTENTION

|swap-file|	关于交换文件的位置和命名
|:preserve|	手工刷新交换文件
|:swapname|	查看原文件及其交换文件的名字
'updatecount'	连续击键多少次后做一次同步
'updatetime'	多长时间之后做一次同步
'swapsync'	同步交换文件时是否同时做一次磁盘同步
'directory'	列出存放交换文件的目录
'maxmem'	尚未写入交换文件的内容所受的内存限制
'maxmemtot'	同上，但是针对所有文件.

下一章：|usr_12.txt| 奇技淫巧
版　权：请参考|manual-copyright| vim:tw=78:ts=8:ft=help:norl:

usr_12.txt Vim 6.3版.最后修改: 2004年05月01日

VIM用户手册--- 作者: Bram Moolenaar

奇技淫巧

将这些看似散兵游勇的命令组合在一起，你将使Vim获得近乎无所不能的威力. 本章将对这些技巧进行一番小小的展示. 其中用到的正是前面章节中介绍的一些命令.

下一章: |usr_20.txt| 快速键入命令
前一章: |usr_11.txt| 灾难恢复
目 录: |usr_toc.txt|

12.1 替换一个word

下面的替换命令将把所有出现的word替换为指定的内容:

ex command
```
:%s/four/4/g
```

"%"这个范围意为对所有行执行该命令. "g"标志则指示替换操作将一行中所有出现的目标字串都进行替换.

不过如果你的文件里面有"thirtyfour"这样的词，结果就不是你想要的了，它将会被替换为"thirty4". 要避免这种例外. 可以在要替换的目标字前面加上"\<"，它匹配一个word的起始位置:

ex command
```
:%s/\<four/4/g
```

[1]译: 仅对Unix类系统有意义

不过显然，碰到"fourty"这样的词还是会弄错．"\>"可以用来指示一个word的结束位置：

```
ex command
:%s/\<four\>/4/g
```

如果你是在写程序，你可能只想替换那些出现在注释中的"four"，代码中的留下． 这可有点为难，"c"标志可以让每个目标被替换之前询问你的意见：

```
ex command
:%s/\<four\>/4/gc
```

替换多个文件中的目标

要对多个文件进行同样的替换操作．显而易见的办法是逐个编辑每个文件，敲入替换命令． 不过用宏记录和回放功能就快多了．

假设你有一个目录下有很多C++文件，都以".cpp"为扩展名． 现欲将所有名为"GetResp"的函数更名为"GetAnswer"．

vim *.cpp	启动Vim，同时指定了要编辑的文件列表：所有的C++文件．现在你正在编辑的将是第一个文件．
qq	开始宏记录，将后续的操作记录在名为q的宏中．
:%s/\<GetResp\>/GetAnswer/g	在第一个文件中执行替换操作．
:wnext	保存该文件并转到下一个文件进行编辑
q	停止宏记录
@q	执行名为q的宏． 它将执行前面记录的替换操作和":wnext． 你可以看看整个过程有没有什么错误．
999@q	对其余文件执行同样的操作[1]

执行到最后一个文件时你会得到一个错误消息，因为":wnext"命令没有文件可以"next"了． 错误会让命令停下来，不过有用的操作已经完成了[2]．

> **备注：** 回放一个宏记录时，任何错误都会导致整个宏记录停止执行． 所以你要确保你在记录宏的过程中没有错误消息．

还有一种例外： 如果其中一个.cpp文件中连一个"GetResp"都没有，替换操作会引起一个错误，整个宏的执行也会被停下来． 标志"e"正是致力于消除这一副作用：

[2]译：对错误的这种处理可以看作是Vim中一个别有用心的设计，它使得一个粗略指定循环次数的操作得以在所有操作对象都被遍历后正确地结束

ex command
```
:%s/\<GetResp\>/GetAnswer/ge
```

"e"标志告诉":substitute"命令就算没找到一个匹配的目标你也不要报错.

12.2 将"Last, First"改为"First Last"

如果你有如下一个名字列表:

Display
```
Doe, John
Smith, Peter
```

现想把它替换为:

Display
```
John Doe
Peter Smith
```

这样的形式. 这样的操作在Vim中只需一个命令:

ex command
```
:%s/\([^,]*\), \(.*\)/\2 \1/
```

我们来肢解这个命令. 显然它是一个替换命令. "%"指定了它的作用范围: 每一行.

替换操作的命令参数形如"/from/to/". 斜杠是用来分隔"from"和"to"的. 下面是本例中"from"部分对应的内容:

Display
```
                  \([^,]*\), \(.*\)

第一个部分位于\( \)之间, 对应 "Last"          \(     \)
匹配除逗号外的任何东西                            [^,]
任意次重复                                      *
匹配 ", "                                      ,
第二个部分位于\( \)之间, 对应 "First"          \(  \)
任意字符                                        .
任意次重复                                      *
```

在对应"to"的部分我们指定了"\2"和"\1". 这在Vim中被称作反向引用. 它们可以用来指代此前在\(\)中匹配的内容. "\2"指代在第二

个"\(\)"中匹配的内容，也就是"First"部分，"\1"则指第一个\(\)中的内容,即"Last" 部分.

你可以使用多达9个的反向引用．"\0"特指整个匹配到内容．除此外在替换命令中还有更多特殊的注意事项．请参考|sub-replace-special|.

12.3 排序

在一个Makefile中通常会有一长串的文件列表，形如:

Display
```
OBJS =  \
        version.o \
        pch.o \
        getopt.o \
        util.o \
        getopt1.o \
        inp.o \
        patch.o \
        backup.o
```

通过一个外部的排序程序sort可以对这一文件列表进行排序:

ex command
```
/^OBJS
j
:.,/^$/-1!sort
```

上面的命令首先会跳转到第一行，即开头是"OBJS"的行，然后再下移一行，用sort程序过滤自该行直至下一个空行．也可以在Visual模式下选择要排序的行然后用"!sort"命令．这样看起来也更容易，不过要排序的行很多时就费劲了.

排序后的结果如下:

Display
```
OBJS = \
        backup.o
        getopt.o \
        getopt1.o \
        inp.o \
        patch.o \
        pch.o \
        util.o \
        version.o \
```

注意每行最后的反斜杠是用来表明该行的内容将在下行延续．但排序后问题出来了！ 本来是最后一行的"backup.o"排序后被置于其它的位置，它缺少一个结尾的反斜杠[1]

最简单的办法是用"A \<Esc>"命令来为它追加一个反斜杠． 排序后最后一行的反斜杠倒是可以保留，不过这样它的下一行必需是一个空行才行．

12.4 反转行序

|:global|命令可以与|:move|命令结合起来将所有行移到第一行的前面，这样的结果将是得到一个各行反序排列的文件．该命令如下：

ex command
```
:global/^/m 0
```

也可简写为：

ex command
```
:g/^/m 0
```

"^"这个正则表达式匹配一行的开头(即使该行是空行也可匹配到)．|:move|命令则将匹配的行移到神秘的第0行之后，所以该行会变成第一行．|:global|命令并不会因这种行号的易序而发生错乱．它将继续处理剩余的行，并将每一行逐个放到文件的第一行去[2]

这对一串连续的行也同样可行． 首先移到第一行并以"mt"标记该行．然后移到最后一行执行命令[3]：

ex command
```
:'t+1,.g/^/m 't
```

12.5 统计字数

有时你会被要求写一篇限定最大字数的文章--- Vim可以自动统计字数．

写到某一阶段时，你可以用：

normal mode command
```
g CTRL-G
```

[1]译：这是Makefile文件规则的要求
[2]译：少数一些命令的执行会改变命令本身的处理，如Join一行
[3]译：借助标记并不是必需的，如：:10,20g/^/m9

来统计一下目前已经写了多少字． 记住不要在"g"命令之后键入空格，这里用空格只是为了命令本身的可读性．

输出结果的形式如下：

Display
```
Col 1 of 0; Line 141 of 157; Word 748 of 774; Byte 4489 of 4976
```

你可以看到当前光标所在的word在整个文章中是第几个(748)，全部的word又有多少个(774)．

如果你要统计整个文件的其中一部分内容，你可以将光标移到要统计部分的开头处执行"g CTRL-G"命令，然后移到要统计部分的末尾再用一次"g CTRL-G"． 计算两次命令得到的当前word位置之差，就得到这部分内容的字数统计．这倒是个不错的练习，不过还有另一种更简单的办法．使用Visual模式，选择要统计的部分． 然后用"g CTRL-G"． 结果会是：

Display
```
Selected 5 of 293 Lines; 70 of 1884 Words; 359 of 10928 Bytes
```

欲知统计字数，行数等其它的统计项，请参考|count-items|．

12.6 查找帮助页[1]

当你在Vim中编辑一个shell脚本或C程序时，你可能会为一个命令或函数需要查找它的帮助页(假设在Unix系统)．首先我们用最简单的方法：将光标移到要查找帮助的关键字上按下：

normal mode command
```
K
```

Vim会对光标所在的词执行"man"程序．找到就显示． 用的是默认的分页程序来处理上下滚动(通常是"more"程序)．到达帮助内容的底部时按<Enter>会让你回到Vim．

不过这种办法的缺点是你不能在编辑的同时看到它． 下面的小技巧会让man页同时显示在一个Vim窗口中，首先运行man对应的文件类型plugin：

ex command
```
:source $VIMRUNTIME/ftplugin/man.vim
```

如果你经常光顾的话就把它放到vimrc文件里． 现在你可以用":Man"[2]命令来打开一个窗口显示man页了：

[1]译：仅对Unix类系统有意义
[2]此处用的Man是一个Vim的命令，不是unix中的man命令，大小写没错

ex command
```
:Man csh
```

你可以在新打开的窗口中任意滚动，其中的文本被进行语法高亮．这样你可以方便地找到你要的帮助信息．"CTRL-W w"会跳转到你正在编辑的内容窗口中．

要查找一个指定小节中的帮助页，可以把节号放要命令前面，如下面的命令查找第3节中的"echo"帮助：

ex command
```
:Man 3 echo
```

要跳转到另一个man页，这种标识一个关键字有一个man页的典型的形式是"word(1)",在该关键字上使用CTRL-]命令． 下次使用":Man"命令时它会在已打开的窗口中显示[1]

要显示当前光标所在字对应的man页，可以用命令：

normal mode command
```
\K
```

(如果你重定义了<Leader>，就用你指定的字符来代替反斜杠)． 比如，你在写下面一行程序时想知道"strstr()"函数的返回值是何类型：

code
```
if (strstr(input, "aap") == )
```

将光标置于"strstr"的某个字符上，按下"\K"． Vim将会打开一个新窗口显示strstr() 对应的man页[2]．

12.7 消除多余空格

很多人都发现行尾的那些空格跳格键又没用又浪费，而且放在文件里显得很不优雅． 下面的命令可以移除所有此类的行尾空白：

ex command
```
:%s/\s\+$//
```

[1] 译：(1)可以在打开的man窗口的任意字上使用CTRL-]命令，而不仅仅是那些以"word(1)" 的形式明确标示为一个man页的关键字，如果光标所在的字并没有一个对应的man页．Vim会显示找不到对应的man页，但这一功能只有在位于man窗口中有效，在另外的窗口中就不行了． (2) 显示man页的缓冲区会被命名为"word.~" 这样的形式．该缓冲区并不会被写回磁盘，除非你自己使用了存盘命令． (3) 使用:vertical Man ls并不能打开一个垂直分隔的窗口，这是一个例外

[2] 译：前提是你已经运行了man.vim，否则这与使用"K"命令效果一样

命令中指定的行号范围是"%"，即应用于每一行中． ":substitute"命令要查找的字串是"\s\+$".这会查找位于行尾的一个或多个空白字符，稍后我们会解释怎样写这些魔咒般的正则表达式|usr_27.txt|.

":substitute"命令的"to"部分是空内容:"//". 即把此类的空白都替换为空，在效果上也就等价于删除了这些空白.

另一种无用的空白是位于制表符前面的空白． 通常删除它们并不影响缩进量． 但也并不绝对！ 所以你最好是手工处理，使用下面的搜索命令:

normal mode command

```
/
```

当然这是皇帝的新衣，你什么也没看到，实际上该命令以一个空格和一个制表符作为要搜索的目标．即"/<Space><Tab>". 现在用"x"命令来删除该空格同时检查一下空白部分有没有改变．如果缩进量改变了的话，你需要再插入一个制表符． "n"命令可以查找下一个此类情况． 如此重复直到处理完整个文件．

12.8 查找一个word在何处被引用

如果你是一个UNIX用户，你就可以将Vim与强大的grep命令结合起来编辑那些包含了某个词的所有文件．如果你在编辑一个程序，希望编辑或浏览所有使用了某个变量的文件．这一功能对此十分有用．

比如，你希望编辑所有包含了"frame_counter"的C文件:

shell command

```
vim \VimBacktick{}grep -l frame_counter *.c\VimBacktick{}
```

我们来细看一下这个命令． grep命令查找一个指定的词． 由于命令中指定了"-l"参数，所以该命令将只是列出包含了该词的文件名而不显示匹配的行． 被查找的目标字符串是"frame_counter". 实际上，此处也可以是任何合法的正则表达式(注意: grep程序的正则表达式并不跟Vim所用的完全相同[1].)

整个命令被反向引用符[2]包围起来． 这个特殊符号告诉UNIX的shell:运行其中的命令，并将命令执行的结果作为当前命令的一部分，就好象是你把这些结果手工在此键入一样． 所以整个命令的结果就是grep命令生成一个文件列表，该文件列表又作为Vim的编辑命令的参数． 你可以用":next"和":first"命令来遍历整个文件列表[3].

[1]译: ≪Mastering the Regular Expression≫一书中列出了不同工具中正则表达式的差别

[2]译: 不要与":substitute"命令中的\1混淆

[3]译: MS-DOS上不能使用该命令的原因并不是它没有grep程序. Cygwin 提供有这样功能的grep. 关键是它的SHELL不支持对`字符类似的解释功能

查找每一行

前面讲述的命令只会找到其中包含了指定word的文件. 通常你也需要知道这些词在文件中的具体位置.

Vim中有一个内置的命令可以用来在一个文件列表中查找一个指定的字符串. 如果你想查找所有C程序中的"error_string", 就可以用下面的命令:

ex command
```
:grep error_string *.c
```

这会使Vim在所有指定的文件(*.c)中查找字符串"error_string". 编辑器也打开匹配的第一个文件并将光标置于第一个包含了"error_string"的行上. 要跳转到下一个匹配的行(不论当前光标位于何处), 使用":cnext"命令即可. 同样, 跳转到前一个匹配的行可以用":cprev"命令, ":clist"命令则可以一次列出所有的匹配.

内置的":grep"命令用到了外部命令grep(Unix)或findstr(Windows). 你也可以通过'grepprg'选项来指定一个执行同样功能的第三方亲信.

下一章: |usr_20.txt| 加速冒号命令

usr_20.txt Vim 6.3版.最后修改: 2003年04月30日

VIM用户手册--- 作者: Bram Moolenaar

加速冒号命令

Vim有一些通用的特性使得命令本身的编辑也十分容易. 冒号命令本身也可以缩写, 可以编辑和重复. 而补齐几乎对任何东西都适用.

下一章: |usr_21.txt| 进退之间
前一章: |usr_12.txt| 奇技淫巧
目　录: |usr_toc.txt|

20.1 命令行编辑

使用冒号命令或者用/或?搜索字符串时, Vim会把光标置于屏幕底部. 命令和要搜索的字串都在这里输入. 该行叫命令行. 注意在这里输入搜索字符串时它也是命令行.

命令行上最显而易见的编辑是使用<BS>键. 它会删除光标之前的字符. 要删除早先输入的光标之前的字符, 你需要首先移动光标键.

比如你输入了如下的命令:

ex command
```
:s/col/pig/
```

按<Enter>键之前, 你想起"col"应该是"cow"才对. 要更正它需要按<Left>5次. 光标现在位于"col"之后了. 按下<BS>和"w"进行改正:

ex command
```
:s/cow/pig/
```

现在你可以按下回车键了. 无需再将光标移至命令行尾部.

在命令行上最常用的一些位移键是:

List
```
<Left>                  向左一个字符
<Right>                 向右一个字符
<S-Left> 或 <C-Left>    向左一个单词
<S-Right> 或 <C-Right>  向右一个单词
CTRL-B 或 <Home>        到命令行行首
CTRL-E 或 <End>         到命令行行尾
```

备注: <S-Left>(按下Shift键的同时按下左箭头键)和<C-Left>(按下CTRL键的同时按下右箭头键)不一定能适用于所有的键盘. 对其它的Shift和CTRL组合键也是一样.

你也可以用鼠标来移动光标.

删除

前面已经说过，退格键<BS>可以删除光标之前的一个字符. 要删除光标之前的整个单词，命令是CTRL-W.

normal mode command
```
/the fine pig

            CTRL-W
/the fine
```

CTRL-U则删除光标之前的所有已键入的内容. 让你可以完全重新开始.

改写

插入键<Insert>可以切换是插入字符还是改写两种编辑模式. 如以下命令:

normal mode command
```
/the fine pig
```

用2次<S-Left>(或者<S-Left>不能用的话，使用左箭头键8次)将光标移到"fine"的开始处. 现在按下插入键<Insert> 切换到改写模式，然后键入"great".

normal mode command
```
/the greatpig
```

哦，天哪，空格给弄没了． 现在也别再用退格键<BS>了，因为它会删除"t"(这跟替换模式还不一样)． 再按一次插入键<Insert>来切换到插入模式，键入空格：

normal mode command
```
/the great pig
```

撤消

一开始你想执行一个冒号命令或搜索一个字串，但中间改变了主意．按下CTRL-C或<Esc> 可以放弃所有已经键入的命令．

> 备注： <Esc>是一个通用的"退出"键．不幸的是，在经典的老Vi上按下<Esc>键却会执行这个冒号命令！ 这可能是一个bug，所以Vim用<Esc>来撤消命令． 不过通过改变'cpoptions'选项Vim还是可以兼容Vi的这种做法． 同时你使用键盘映射时(可能是为Vi而定制的键盘映射)<Esc>键也将兼容Vi的做法.所以使用CTRL-C是通用的好办法[a].)
>
> [a]译者： 至少有一个例外： 按下CTRL-W开始一个窗口相关的操作时，只能用<Esc>来撤消整个命令，因为CTRL-W CTRL-C是关闭当前窗口的命令

如果光标位于冒号命令的行首，按下<BS>键会撤消该命令． 就好象把":"或"/"删除了一样．

20.2 命令行缩写

有一些冒号命令实在是太长了． 前面已经提过":substitute"命令可以缩写为":s"． Vim中这是一个普适的规则，所有的冒号命令都可以被缩写．

一个命令最短能有几个字符？ 字母一共也就26个，很多命令比如":set"也是以":s"开头,但":s"并不是指":set"， ":set" 对应的缩写是":se"．

一个缩写形式同时可以对应两个命令时，它只能指代其中的一个．到底是哪一个？ 这里可没有什么诀窍，你还是得自己good good study． 在帮助文件中会提到一个命令的最短的缩写形式，如：

Display
```
:s[ubstitute]
```

这意味着":substitute"最短的缩写形式是":s"． 其后的字符都是可有可无的． ":su"和":sub"也同样可以．

在用户手册中我们要么用命令的全名，要么是用仍具可读性的缩写．比如":function"可以被缩写为":fu"． 但是多数人看不明白它是哪个单词的缩影，所以我们会用":fun"． (Vim没有一个":funny"命令，要不然就算":fun"也会引起冲突呢)．

在写Vim脚本时建议大家用命令的全名． 这样什么时候回头读这些脚本时就更方便了解它的意思． 除非是一些象":w"(":write")和":r"(":read")一样太过常用的命令．

特别容易混淆的是":end"，可以代表":endif"，也可以代表":endwhile"或"endfunction所以最好全部用它们的全称．

选项的缩写

在用户手册中选项名都用的是它的全名． 很多选项都有一个相应的缩写名． 跟冒号命令不一样，缩写名只是是固定的一个写法，比如，'autoindent'缩写为'ai'． 这两个选项效果完全一样：

normal mode command
```
:set autoindent
:set ai
```

你可以在|option-list|找到一个选项名缩写的完整列表．

20.3 命令行补齐

这个特性是很多人从Vi转到Vim的原因．这是一个格外诱惑人的功能，一旦你用过它就再也离不了了[1]．

假设你有一个目录下有下面3个文件：

List
```
info.txt
intro.txt
bodyofthepaper.txt
```

要编辑最后一个文件：

ex command
```
:edit bodyofthepaper.txt
```

想不出错真是太难了． 更快的办法是：

[1]译：恐怕还会爆出类似这样的感叹： "天哪，为什么让我吃到这么好的一顿饭，我以后吃不到该怎么办 !" --<<食神>>评委薛家燕喜忧参半地嘶吼道

ex command
```
:edit b<Tab>
```

效果是一样的，怎么回事？ 制表符<Tab>键补全了光标之后的单词.此例中该单词是"b". Vim会在该目录中查看哪个文件是以"b"开头的.正是你要找的那个文件，所以Vim为你补全了整个文件名.

现在键入命令：

ex command
```
:edit i<Tab>
```

Vim会报以一声蜂鸣，告诉你：

ex command
```
:edit info.txt
```

蜂鸣声是告诉你Vim找到符合的文件可不止一个. 它先用第一个(按字母顺序). 如果你再按一下制表符<Tab>键，文件变为：

ex command
```
:edit intro.txt
```

所以，如果一个<Tab>补全的文件名不是你想要的，那就再按一次，有多个符合的文件名时,你可以一次一个把它们都过一遍.

最后一个后再按下<Tab>键又会回到第一个去：

ex command
```
:edit i
```

它会从头再来一遍. Vim会在整个列表中轮番循环，CTRL-P可以反序遍历整个列表.

Display
```
      <-------------------- <Tab> -------------------------+

          <Tab> -->                  <Tab> -->
:edit i        :edit info.txt      :edit intro.txt
          <-- CTRL-P                  <-- CTRL-P
      |
      +---------------------- CTRL-P ---------------------->
```

上下文

现在不用":edit i"，你用命令":set i"然后按下制表符键<Tab>试试：

ex command
```
:set icon
```

呵，你怎么没看到":set info.txt"? 这是因为Vim进行补全时考虑到了命令的上下文环境. 用什么补全要依命令而定. Vim知道你不会在":set"命令后面跟一个文件名，很明显你要的是一个选项.

再重复按制表符<Tab>键，Vim会轮循所有的匹配. 可不少呢，所以最好是多输入几个字符再让它进行补全:

ex command
```
:set isk<Tab>
```

这次补全出来的是:

ex command
```
:set iskeyword
```

现在输入"="再按制表符键<Tab>:

ex command
```
:set iskeyword=@,48-57,_,192-255
```

Vim插入了该选项目前的设置，所以你可以直接在这些修改了.

Vim会在<Tab>之后补全它当下想要的东西. 亲自去试一下你就会知道它是怎么回事.有些情况下补全功能也不能如你所愿，这是因为要么Vim不知道你到底想要什么，要么此时的补全功能还没实现. 这种情况下你会看到一个真正的制表符被放在当前位置(以^I显示).

列出所有匹配

有众多的补全候选项时，最好是看一下它全貌: 用CTRL-D. 比如在下面的命令之后按CTRL-D:

ex command
```
:set is
```

结果是:

ex command
```
:set is
incsearch  isfname    isident    iskeyword  isprint
:set is
```

Vim给出了整个列表. 现在一切都对你一览无余了. 如果你要的东西不在其中, 还可以用退格键<BS>来修改已输入的单词. 如果你觉得整个列表太大了, 可以多输入几个字符缩小包围圈然后用制表符键<Tab>重新让它补全.

如果你够细心的话, 你会发现上面的"incsearch"并不以"is"开头. 这种情况下"is"匹配的是它的缩写形式. (很多选项都同时有缩写形式) Vim很机灵的, 它会猜到你也许是想用把缩写的选项名扩展为长的全名形式.

更多内容

CTRL-L命令会最大限度地补全各候选项的共同部分. 如果你输入了":edit i", 同时有两个候选项"info.txt"和"info_backup.txt", 它就会补全为":edit info".

'wildmode'选项可以调整补全的工作方式.
'wildmenu'选项使候选项以类似菜单的形式出现.
'suffixes'选项可以指定哪些候选项不太重要, 可以放在列表的最后.
'wildignore'选项指定一些要排队在候选项列表之外的项[1].

更多细节请参考|cmdline-completion|

20.4 命令行历史记录

在第3章简要提到了历史记录. 基本的思想是可以用上下箭头键来找回用过的命令.

实际上有4个历史记录. 这里将要提到的是冒号命令历史记录和"/"及"?"的搜索命令历史记录, "/"和"?"都是搜索命令, 所以它们共享同一个历史记录. 另外的两类历史记录分别是关于表达式和input()函数的输入内容的. 请参考|cmdline-history|

假设你用过一次":set"命令,接着用过10次其它的冒号命令之后又想重复这个":set"命令. 你可以按下":"然后按10次上箭头键<Up>. 更快的办法是:

ex command
```
:se<Up>
```

Vim会回到上一次你以"se"开头的命令去. 这样你离":set"命令就更近一些了. 至少你不用按10次上箭头键<Up>了(除非这中间的10个冒号也都是":set"命令).

[1]译者: (1)Vim选项众多, 但多数是一小簇相互关联的选项针对某功能或某核心命令进行修饰, 调整(2)Vim不会给出一个粗糙难用的功能, 它会用各种选项来微调这项功能.使它真正贴近实用, 这里对补全这一功能的精雕细琢就是一例

上箭头键<Up>会根据目前键入的命令部分去跟历史记录进行比较. 只有符合的才会被列出来.

如果没找到，你还可以用下箭头键<Down>回到刚才输入的部分命令进行修改. 或者用CTRL-U命令全部删除后重来.

要查看所有的历史记录，用命令:

ex command
```
:history
```

列出的是冒号的历史记录. 要查看搜索命令的历史记录，用[1]:

ex command
```
:history /
```

CTRL-P效果如同<Up>，唯一的不同是它不会根据你已经键入的部分去遍历记录. 类似地，<Down>的对应物是CTRL-N和下箭头键. CTRL-P代表previous，CTRL-N代表Next [2]

20.5 命令行窗口

[3] 命令行上的编辑跟普通的文本内容编辑不太一样. 没有那么多的命令可用. 对多数命令来说这也不是什么大问题. 但有些复杂的命令就不一样了. 这时命令行窗口可就用处大了.

下面的命令可以打开命令行窗口[4]:

normal mode command
```
q:
```

Vim在屏幕底部打开了一个(小)窗口. 该窗口的内容是历史记录，最后一行是空行:

[1]译者: 规律:跟map，autocommand，tags等这些东西一样，以一个列表或集合为操作对象的命令都有一个命令列出整个列表，也有遍历它们的方法

[2]译者: 规律，CTRL-P和CTRL-N经常用于上下遍历一个列表，如shell的命令行历史. emacs中上下跳转一行，进行各种补齐，以及此处的遍历历史记录

[3]译者: 规律: 为了浏览或编辑的方便起见，Vim往往有一个办法将一个列表或集合操作的对象放到一个窗口中去

[4]译者: 注意这是在normal模式下键入的命令

Display
```
+---------------------------------------+
|other window                           |
|~                                      |
|file.txt===============================|
|:e c                                   |
|:e config.h.in                         |
|:set path=.,/usr/include,,             |
|:set iskeyword=@,48-57,_,192-255       |
|:set is                                |
|:q                                     |
|:                                      |
|command-line===========================|
|                                       |
+---------------------------------------+
```

现在是Normal模式. 可以用"hjkl"四处移动. 比如用"5k"命令上移到":e config.h.in" . "$h"到单词"in"的"i"上, 键入"cwout". 现在该行变为了:

ex command
```
:e config.h.out
```

现在按下回车键该命令就会被执行. 而命令行窗口也同时会被关闭.

回车键会执行当前行的命令. 不管Vim身处Insert模式还是Normal模式.

对命令行窗口作出的修改不会被保存. 历史记录不会因此被改写. 除非你执行的命令是被追加到历史记录中, 就象其它被执行过的命令一样[1]

命令历史记录窗口有时十分有用, 你可以在此浏览整个历史记录, 找到一个相近的命令, 稍加修改, 然后重新执行它. 也可以用搜索命令来进行查找.

在上例中"?config"命令就可以查找包含"config"的命令. 看起来很奇特, 你在用一个命令行命令来查找命令行窗口中的另一个命令. 键入这个命令行命令时你就不能再打开另一个命令历史记录窗口了, 它只能有一个[2]

[1]译者: 历史就是这样, 不能被改写, 只能被延续. 除非你有月光宝盒. :-).
[2]译者: 用CTRL-W c命令来关闭命令历史记录窗口时, Vim会在命令行上显示当前行的命令, 此时若不想执行该命令, 可以按下CTRL-C或<Esc>, 窗口同时会被关闭, 否则按下回车的话, 该行的命令还是会被执行, 而用:q退出该窗口时就不会这样

下一章：	|usr_21.txt| Go away and come back
版　权：	请参考|manual-copyright| vim:tw=78:ts=8:ft=help:norl:

usr_21.txt Vim 6.3版.最后修改: 2004年03月29日

VIM用户手册--- 作者: Bram Moolenaar

进退之间

彼节者有间而刀刃者无厚，以无厚入有间，恢恢乎其于游刃必有余地矣。

---庄子≪养生主≫

本章的内容是关于如何使Vim和其它程序的结合运用. 比如在Vim中执行一个外部程序，然后再回到Vim. 另外，还有关于如何记下Vim的当前状态，稍后再恢复到该状态.

下一章: |usr_22.txt| 查找文件
前一章: |usr_20.txt| 加速冒号命令
目 录: |usr_toc.txt|

21.1 挂起与恢复

象其它的Unix程序一样Vim也可以用CTRL-Z来挂起. 该命令停止Vim的执行使你回到启动Vim的shell中去. 你可以做些别的事. 完了再用"fg"命令回到Vim.

shell command
```
CTRL-Z
{执行任何shell命令}
fg
```

此时你会准确地回到你离开时的状态，所有东西都还是原封不动.

如果CTRL-Z不行，你还可以用":suspend"命令. 别忘了等会再让回到Vim，否则你会丢失所有的改动.

只有Unix系统支持该操作. 其它系统上Vim会启动一个新的shell. 这同样可以让你在其中执行别的shell命令, 不过它是一个新的shell, 不是你启动Vim的那个[1].

运行GUI版本的Vim时你不会回到启动Vim的shell去. CTRL-Z等效于将窗口最小化[2]

21.2 执行shell命令

如果只执行一个shell命令的话可以在Vim中用":!{command}"命令. 比如, 要查看当前目录的内容:

ex command
```
:!ls
:!dir
```

第一个命令用于Unix, 第二个用于MS-Windows.

Vim会执行指定的程序. 程序结束时它会提示你按回车键继续. 这样可以让你有机会看一下程序的输出是什么.

"!"也被用在其它几个需要运行程序的地方. 我们来巡视一下此类命令:

ex command

:!{program}	执行 {program}
:r !{program}	执行 {program}并读取它的输出
:w !{program}	执行 {program}并把当前缓冲区的内容作为它的输入
:[range]!{program}	以{program}过滤指定的行

[3]注意在"!{program}"前面指定一个行的范围可就大不相同了. 没有的话就是象通常一样执行该程序, 带上范围则会把指定的行送入该程序进行过滤.

以这种方式执行一组程序也可以, 不过另开一个shell是最好的选择. 你可以这样打开一个新的shell:

ex command
```
:shell
```

[1]译者: 这在有些情况下还是略有区别, 每个shell还是有一些不同之处的, 比如变量

[2]译者: 在windows下的gvim中, CTRL-Z通常被映射为撤消命令u, 要看到它的窗口最小化效果, 先执行命令:unmap <CTRL-Z>删除这个映射, 然后再按CTRL-Z

[3]译者: 如何向外部程序传递某些行, 同时又不伤及这些行的内容, 还要读取该程序的输出?

这类似于用CTRL-Z挂起Vim. 不同之处在于这是一个新的shell.

使用GUI版本的Vim时shell会作为Vim窗口的输入和输出. 因为Vim不是一个终端仿真程序. 这样的配合并非十全十美. 如果你碰到什么问题, 试一下切换'guipty'选项的效果. 如果这样还不行, 可以新开一个终端来运行shell. 比如这样:

ex command
```
:!xterm&
```

21.3 记住编辑信息: viminfo

一番辛苦之后你的寄存器里已经放入了要交换的内容, 也标记了文件中的一些位置, 命令行历史列表中都是你精心锤炼过的命令. 可是一旦你退出了Vim这些东西就会杳然无踪. 好在我们有下面的对策!

viminfo正是被用来存储这些信息的:

List

- 命令行历史记录和搜索命令历史记录
- 寄存器中的内容
- 文件中的标记
- 缓冲区列表
- 全局变量

每次你退出Vim时这些信息都会被存储在viminfo文件中. 下次启动时Vim会读取该文件的内容并据此恢复这些信息.

'viminfo'选项的默认设置可以存储有限的几类信息. 你可能想让它保存更多的东西. 该命令的使用是这样的:

ex command
```
:set viminfo=string
```

其中的string指代你要保存的东西. 其语法是一个选项字符跟一个参数. 一对对的选项/参数以逗号分隔.

看看下面的例子中是如何定义一个viminfo的string的. 首先'选项指你要为多少文件保存标记信息(a-z). 你可以为它先一个适当的偶数作为参数(比如1000). 命令如下:

ex command
```
:set viminfo='1000
```

选项字符f控制是否保存全局的标记(A-Z和0-9). 如果以0为参数则不会存储这些全局标记. 如果参数是1或没有指定f选项, 则会保存. 需要的话你可以这样定制:

ex command
```
:set viminfo='1000,f1
```

选项字符<控制每个寄存器保存多少行. 默认情况下保存所有行. 如果是0, 则不保存. 为了避免你的viminfo有成千行的内容(很可能根本不会用上, 又让你的Vim启动变慢)你可以指定一个行数如500:

ex command
```
:set viminfo='1000,f1,<500
```

其它几个可用的选项字符是:

List

:	要保存的冒号命令历史记录数
@	要保存的输入历史记录数
/	要保存的搜索命令历史记录数
r	可移动介质, 对此不会保存其标记信息(可以多次使用)
!	以大写字母开始并且不含有任何小写字母的全局变量
h	开始时禁用'hlsearch'选项
%	缓冲区列表(只有没指定文件参数启动Vim时才会恢复这些信息)
c	使用'encoding'选项中的方法进行字符编码转换
n	viminfo文件的名字(必需是最后一个选项)

请参考'viminfo'选项的帮助和|viminfo-file|了解更多信息.

多次使用Vim后, 只有最后一次退出的Vim会保存这些信息. 这可能会引起前面保存的信息的丢失, 每个对应的项只能保存一次.

回到你上次退出时的位置

如果你的文件只写了一半但圣诞节已经到了. 尽管退出Vim去加入圣诞的狂欢吧, 几个星期后再次启动Vim, 只需键入命令:

normal mode command
```
'0
```

你就会准确地回到你上次退出时的位置, 继续干吧.

每次退出Vim时它都会创建一个特殊的标记. 最后的一个是'0. 上次的'0现在会变成'1, 原来的'1成了'2, 如此类推, 原来的'9就丢掉了.

":marks"命令可用来找出这些从'0到'9的标记都指向哪里.

让VIM共享viminfo

你可以在Vim中用":wviminfo"和":rviminfo"命令来保存和恢复这些信息. 需要在不同的Vim实例中交换信息时这两个命令用处就大了, 比如, 在第一个Vim 保存:

ex command
```
:wviminfo! ~/tmp/viminfo
```

在另一个中读取:

ex command
```
:rviminfo! ~/tmp/viminfo
```

显而易见, "w"是指"write", "r"指"read".

":wviminfo"命令中的!字符可以覆盖原来的viminfo文件. 如果没有这个字符的话目前的信息会追加到原来的viminfo文件后面.

":rviminfo"命令中的!则是说以指定的viminfo中的内容覆盖当前的所有信息, 否则的话只有尚未设置的内容会被追加[1].

这些命令也可以用来保存一个info以备后用. 你甚至可以建一个专门存放viminfo文件的目录, 每个文件一个用途.

21.4 会话[2]

假设你已在Vim中干了一整天了. 现在想退出来第二天接着干. 这时可以把目前的状态以一个会话保存起来, 第二天可以据此恢复原样.

一个Vim的会话含有你编辑活动的[3]所有信息.包括文件列表, 窗口布局, 全局变量, 选项和其它信息. (确切说到底包括什么要看'sessionoptions'选项怎么设置, 见下文)

下面的命令可以创建一个会话文件:

ex command
```
:mksession vimbook.vim
```

稍后如果你想恢复到此前的会话状态, 可以执行命令:

[1]译者: 背反律: 注意这里缺少!并不会给出警告信息说已经有一个同名文件存在了
[2]译者: 会话可以看作是信息更为丰富的viminfo高保真版本
[3]译注: 几乎所有, 并非任何一个编辑状态都会记录

ex command
```
:source vimbook.vim
```

如果你想启动Vim时恢复某个会话，可以直接使用命令行:

shell command
```
vim -S vimbook.vim
```

这告诉Vim启动时读取一个特殊文件. 'S'代表session(实际上，你可以用-S选项来执行任何一个Vim脚本. 从这个意义上说它代表"source").

上次的窗口被恢复，同样的位置，同样的大小. 连映射和各选项的设置也完全一样.

具体会恢复哪些东西要看'sessionoptions'选项的设置. 默认值是"blank,buffers,curdir,folds,help,options,winsize".

List

blank	空窗口
buffers	所有的缓冲区，而不仅是当前窗口中的
curdir	当前目录
folds	folds，包括手工创建的
help	帮助窗口
options	所以的选项和键映射
winsize	窗口大小

你可以根据自己的喜好进行定制. 比如恢复Vim窗口的大小，可以这样:

ex command
```
:set sessionoptions+=resize
```

此会话，彼会话

会话最显见的用途是工作于不同的项目的情形. 假设你把你的会话文件都保存在"~/.vim" 目录下. 当前在"screen"项目，要切换到"boring"项目:

ex command
```
:wall
:mksession! ~/.vim/secret.vim
:source ~/.vim/boring.vim
```

首先用":wall"命令保存所有被改动的文件．然后用":mksession!"保存当前的会话． 这会覆盖掉前面的会话文件．下次你再进入"secret"会话时就可以从目前的位置继续．最后切换到新的"boring"会话．

如果你打开了帮助窗口，分隔了一个，再关闭一些窗口，三番五次后屏幕上可能看起来已经乱七八糟，此时还可以用上次保存的会话文件来恢复：

ex command
```
:source ~/.vim/boring.vim
```

借此你可以控制下次进入该项目时的起始位置，是两次保存目前的改动，还是继续把原来的会话文件作为一个起点．

会话的另一个用法是把你喜好的窗口布局保存为一个会话，下次用会话文件恢复．比如，下面的窗口布局：

Display
```
+----------------------------------------+
|                  VIM - main help file  |
|                                        |
|Move around:  Use the cursor keys, or "h|
|help.txt================================|
|explorer   |                            |
|dir        |~                           |
|dir        |~                           |
|file       |~                           |
|file       |~                           |
|file       |~                           |
|file       |~                           |
|~/=========|[No File]===================|
|                                        |
+----------------------------------------+
```

这个布局中顶部是一个帮助窗口，你可以查看帮助．左边窄窄的垂直窗口是一个文件浏览窗口． 这是一个列出当前目录内容的plugin． 你可以选择要编辑的文件．下一章还会详细讲述这部分内容．

可以从一个刚启动的Vim中建立布局：

ex command
```
:help
CTRL-W w
:vertical split ~/
```

根据个人喜好调整窗口大小． 然后保存该会话：

ex command
```
:mksession ~/.vim/mine.vim
```

现在你可以这样来启动Vim:

ex command
```
vim -S ~/.vim/mine.vim
```

提示: 要打开文件浏览窗中的某文件，只需将光标移到文件名上按"O"，或用鼠标双击该文件.

UNIX和MS-WINDOWS

有些人同时使用MS-Windows和Unix. 如果你正是这种情况，你可以考虑把"slash"和"unix"加到'sessionoptions'选项中. 这样会话文件会以兼容于两类系统的形式保存. 命令如下:

ex command
```
:set sessionoptions+=unix,slash
```

Vim会用Unix格式保存，因为MS-Windows 下的Vim能读写Unix文件. 但Unix下的Vim就不能读取MS-Windows格式的会话文件了. 同样地，MS-Windows的Vim能理解以/分隔的文件路径名，Unix下的Vim则不能理解反斜杠\.

会话和VIMINFO

会话能保存很多信息，但是不包括标记，寄存器和命令历史记录. 要用这些东西你还是得靠viminfo.

多数情况下都是独立使用会话和viminfo. 尤其是切换到另一会话但是要保存命令行的历史记录. 这样你可以在一个会话中复制文件，在另一个会话中进行粘贴.

你也可以同时使用会话和viminfo. 不过要自己动手. 如:

ex command
```
:mksession! ~/.vim/secret.vim
:wviminfo! ~/.vim/secret.viminfo
```

恢复时是这样:

ex command
```
:source ~/.vim/secret.vim
:rviminfo! ~/.vim/secret.viminfo
```

21.5 视图

会话保存了整个Vim的外观. 但如果你只想保持一个窗口的属性的话，用视图就行了.

视图用于这样的情况: 你想以一种特殊的方式编辑该文件. 比如打开'number'选项，定义一些folds. 就象会话一样，你可以保存视图稍后再恢复它. 事实上，保存一个会话时它也的确保存了每个窗口的视图.

视图有两个基本用法. 第一个是让Vim决定视图文件. 你可以在后来编辑该文件时恢复该视图. 要保存当前窗口的视图:

ex command
```
:mkview
```

Vim会决定如何保存视图. 下次你再编辑该文件时你可以用下面的命令让它自己载入上次保存的视图[1]:

ex command
```
:loadview
```

很简单，是不是?

如果你想在浏览文件时关闭'number'选项，或者打开所有的folds，你可以改变这些选项的设置来调整你的观感. 然后这样保存该视图:

ex command
```
:mkview 1
```

当然，你可以这样恢复它:

ex command
```
:loadview 1
```

现在你可以通过":loadview"带不带参数来切换两个不同的视图了.

用这种办法一共可以保存10个视图，从1到9再加上一个不带数字的视图.

[1]译者: 实际上Vim会将一个VIM脚本保存在选项'viewdir'所指定的目录下. 该脚本的名字以一种特定方式编码，如对于编缉E:\work\tips\vim.tip文件时执行的:mkview，保存的文件名是"E=-=+work=+tips=+vim.tip="，这样执行:loadview时Vim就可以根据这种约定的编码来查看当前被编辑的文件是否有对应的view脚本文件

指定文件名的视图

第二种方式是保存视图时指定一个文件名． 这个视图也可以在编辑别的文件时载入． Vim 会自动切换到该视图中指定的文件去． 这样你可以方便地切换到另一个文件进行编辑，同时相应的选项设置也可随身携带．

比如你保存了当前文件的视图：

ex command
```
:mkview ~/.vim/main.vim
```

可以这样恢复它：

ex command
```
:source ~/.vim/main.vim
```

21.6 模式行

编辑某个特殊文件时，你可能希望为该文件特别一些选项．每次输入这些命令实在是太折磨人了． 使用会话或视图的话又不便于让每个人都去共享这些设置．

解决之道是为每个文件添加一个模式行． 这是一行让Vim设置指定选项的文本，只对当前文件有效．

典型的例子是C程序中，你想把缩进量设为4个空格．这需要把'shiftwidth'选项设置为4． 相应的模式行是：

Display
```
/* vim:set shiftwidth=4: */
```

把这行文本放到文件的前5行中或最后5行中． 编辑该文件时，你看看'shiftwidth'选项是不是已经设置为了4． 编辑其它文件时，它又会回到默认的设置:8．

对一些文件来说把模式行作为文件头挺好，这样模式行就可以放在文件开头．对于可能打乱正文的情况来说，可以把它放在文件尾．

'modelines'选项控制Vim前后检查多少行来找出模式行． 比如检查10行：

ex command
```
:set modelines=10
```

'modeline'选项可以关闭这项功能. 如果你身为root用户, 编辑的文件时战战兢兢如履薄冰那最好还是关闭它:

ex command
```
:set nomodeline
```

模式行的格式是这样的:

code
```
any-text vim:set {option}={value} ... : any-text
```

其中的"any-text"是说你可以在对Vim有用的部分文本之前或之后放入任何东西. 这使得模式行可以以一个注释的形式出现, 比如上例中的/*和*/.

Vim根据" vim:"来识别模式行. "vim"之前必需有一个空格, 或者"vim"位于行首. 这样你用"gvim:"的话就不行.

冒号之间的是":set"命令. 除了你需要在作为选项内容的冒号前面放一个反斜杠之外. 它跟直接用":set"命令没有两样,

看另一个例子:

code
```
// vim:set textwidth=72 dir=c\:\tmp:  use c:\tmp here
```

第一个冒号之前有一个反斜杠, 这样它才可以作为":set"命令的内容. 第二个冒号的内容会被忽略, 这样在此加上一段注释也不错.

关于模式行的更多内容请参考|modeline|.

下一章: |usr_22.txt| 查找文件

usr_22.txt Vim 6.3版.最后修改: 2003年03月17日

VIM用户手册--- 作者: Bram Moolenaar

查找文件

处处都是文件. 到哪去找你要编辑的文件呢? Vim提供了几种浏览目录的方法. 还有的命令是根据在一个文件中的信息去判断要编辑的其它文件. Vim也会跟踪编辑过的文件.

下一章: |usr_23.txt| 编辑其它文件
前一章: |usr_21.txt| 进退之间
目 录: |usr_toc.txt|

22.1 文件浏览器

Vim有一个可以编辑目录的插件. 试一下:

ex command
```
:edit .
```

在自动命令和Vim脚本的魔力下, 显示出来的窗口中将是当前目录下的内容. 如下:

Display
```
" Press ? for keyboard shortcuts
" Sorted by name (.bak,~,.o,.h,.info,.swp,.obj,.orig,.rej at end of list)
"= /home/mool/vim/vim6/runtime/doc/
../
check/
Makefile
autocmd.txt
change.txt
eval.txt~
filetype.txt~
help.txt.info
```

你可以在其中看到这样的条目：

Display

```
1.   一个关于用?命令来获取如何使用文件浏览器的注释.
2.   第二行指示其中的文件列表以何种顺序进行排列. 它们可以以不同的方式进行排序.
3.   第三行是当前目录名.
4.   "../"目录项. 代表当前目录的父目录.
5.   目录名
6.   普通文件名. 就象第二行中的提示, 有些文件不进行排序而是列在整个列表的末尾.
7.   很少会用的的文件名. 一般来说这些文件很少会被使用, 所以被放在整个列表的末尾.
```

如果你打开了语法高亮，上面提到的不同条目将会以不同的颜色显示. 看起来更易于识别.

你可以在这个特殊的缓冲区中使用Normal模式下的Vim命令来移动. 比如，将光标移到一个文件上按<Enter>. 这将会打开该文件进行编辑. 要回到刚才的文件浏览器. 再次用":edit ."即可，或者，用CTRL-O也一样.

试一下光标停在一个目录名上时按<Enter>会怎么样. 结果是文件浏览器打开该目录并显示其中的内容. 在"../"上按下<Enter>则会上溯到当前目录的父目录. "-"命令也是殊途同归，而且使用"-"命令时不限于当前光标位于何处.

你可以在文件浏览器窗口中使用?命令得到一个简短的帮助说明. 如下[1]:

Display

```
" <enter> : 打开一个文件或目录~
" o : 为文件或目录打开一个新的窗口~
" O : 在前一个被访问过的窗口中打开指定的文件/目录
" p : 预览文件/目录~
" i : 切换大小/日期的列表显示~
" s : 选择一个排序键        r : 逆向排序~
" - : 进入父目录            c : 进入目录
" R : 文件重命名            D : 删除文件
" :help help-explorer得到更多信息~
```

首先要提到的命令是选择显示一个文件的命令. 打开文件的位置因具体命令的不同而不同.

Display

```
<Enter>   使用当前的窗口
o         打开一个新窗口
O         使用最近使用过的其它窗口[[译: 如果只有当前窗口就打开新窗口]]
p         同O, 但同时还将光标定位于该窗口中
          参考 |preview-window|
```

[1]译者: 上面标为"- : 进入父目录"的这一特性不能正常使用

下面的命令用于显示其它信息[1]:

Display

i	切换显示文件的大小和日期/时间信息
s	光标所在的条目作为排序键进行排序. 首先用i命令打开对文件大小和日期的显示. 然后将光标移到任一文件的大小上按下s. 此时所有的文件都将按其大小进行排序. 光标置于日期上时按s则会按其日期进行排序.
r	逆向排序当前的排序列表(either size or date)

此外还有其它一些命令:

List

c	将当前目录切换到当前光标所在的目录名. 接下来使用":edit"命令时就无需附加路径名了
R	更改当前光标下的文件, Vim会提示你输入一个新的文件名[[译: 不支持目录更名]].
D	删除当前光标下的文件名. Vim也会提示你进行确认[[译: 不 支持目录的删除]].

22.2 当前目录

象shell一样, Vim也有一个工作目录的概念. 假设你的工作目录是home目录, 现欲编辑"VeryLongFileName"目录下的几个文件, 你需要这样:

ex command
```
:edit VeryLongFileName/file1.txt
:edit VeryLongFileName/file2.txt
:edit VeryLongFileName/file3.txt
```

为避免每次都键入这个臭长的目录名, 你也可以:

ex command
```
:cd VeryLongFileName
:edit file1.txt
:edit file2.txt
:edit file3.txt
```

":cd"命令会将工作目录指定为"VeryLongFileName". 你可以用":pwd"命令来获得目前的工作目录是什么[2].

[1]译: 最后一项显示的是什么? 不一定按指定条目进行排序
[2]译: pwd意为present working directory, 而不是password

ex command
```
:pwd
/home/Bram/VeryLongFileName
```

Vim会记住你上一次的工作目录，"cd -"命令会两次设定上次工作目录为当前工作目录． 如:

ex command
```
:pwd
/home/Bram/VeryLongFileName
:cd /etc
:pwd
/etc
:cd -
:pwd
/home/Bram/VeryLongFileName
:cd -
:pwd
/etc
```

局部于窗口的工作目录

分隔出一个新窗口时，两个窗口都会使用同样的工作目录． 如果你想在新窗口中编辑其它目录下的一个文件，你也可以为它设定另外的工作目录，同时又不另一窗口中的工作目录． 这在Vim中叫local directory.

ex command
```
:pwd
/home/Bram/VeryLongFileName
:split
:lcd /etc
:pwd
/etc
CTRL-W w
:pwd
/home/Bram/VeryLongFileName
```

目前为止我们还没有用过":lcd"命令． 所有的窗口都共享同一个工作目录． 一旦在其中一个窗口中用":cd"命令改变了工作目录，其它窗口中的工作目录也将随之改变．

对一个窗口使用":lcd"后它的工作目录会被记录下来，这样其它窗口中的":cd"或":lcd"命令就不会再影响到该目录了．

在一个窗口中使用":cd"命令会重设它的工作目录为共享的工作目录．

22.3 查找一个文件

假设你正在编辑一个C源程序，其中有这样一行：

code
```
#include "inits.h"
```

假设你想知道头文件"inits.h"中的内容. 只需将光标置于该文件上然后键入：

normal mode command
```
gf
```

Vim就会找到并编辑该文件.

如果这个文件并不在工作目录下呢？此时Vim会使用在'path'选项中定义的目录去查找它. 该选项的内容是一个以逗号分隔的目录名列表.

假设你的头文件都位于"c:/prog/include". 下面的命令会将该目录加入你的搜索路径中去：

ex command
```
:set path+=c:/prog/include
```

这个目录是一个绝对路径. 所以不管你当前的工作目录为何，它都代表同样的位置. 如果你的头文件是位于一个子目录呢？此时你可以指定一个相对路径. 它以一个点号开始

ex command
```
:set path+=./proto
```

这告诉Vim在你使用"gf"命令的那个文件所在目录的子目录"proto"中查找指定的文件.这样命令的结果是Vim会从该目录开始，查找文件"proto/inits.h"

如果没有指定"./"只有一个"proto"的话，Vim就会在当前的工作目录中查找其下名为"proto"的子目录. 当前工作目录并不一定就是你正在编辑的文件所在的目录.

'path'选项可以以多种方法指定要在哪些目录搜索文件. 请参考关于'path'选项的帮助.

'isfname'选项用于告诉Vim哪些字符可以作为文件名的一部分，哪些不行(比如上例中的"字符).

如果你要找的文件名并没有以出现在当前文件中，但你已经确知它的名字，你可以这样查找它：

ex command

```
:find inits.h
```

Vim会使用'path'中的选项来搜索该文件. 除了'path'选项对搜索路径的影响外, 这与直接使用":edit"命令毫无二致.

要在另一窗口中打开指定的文件的话, 可以以"CTRL-W f" 替代"gf"命令, 或者用":sfind"命令来替代":find"命令[1].

下面的例子可以方便地编辑一个位于'path'中的文件:

shell command

```
vim "+find stdio.h"
```

该命令会在由'path'指定的路径中搜索文件"stdio.h". 其中的双引号是必需的, 它确保引号中的内容作为一个参数传给应用程序. 参考|-+c|

22.4 缓冲区列表

Vim编辑器使用缓冲区这个词来描述被编辑的文件. 事实上, 一个缓冲区是一个被编辑文件的副本. 通常你会在完成对一个缓冲区的编辑后保存该文件. 缓冲区不仅包含了文件的内容, 它也记录了该缓冲区中所有的标记, 设置以及其它跟缓冲区有关的东西.

隐藏缓冲区

假设你正在编辑文件"one.txt"现在需要转而编辑"two.txt". 你可能会直接使用":edit two.txt"命令, 但是你已经对"one.txt"作出了改动, 所以这一命令会失败, 同时你又不希望现在就保存文件"one.txt"的内容. Vim对此的解决方案是:

ex command

```
:hide edit two.txt
```

缓冲区"one.txt"从屏幕上消失, 但Vim保存了它的当前状态. 这叫做隐藏缓冲区: 缓冲区中确有内容但你看不到它.

":hide"命令的参数是另一个命令. 它使该命令工作于'hidden' 选项被设置的状态. 你也可以自行设置该选项. 其效果是当你的缓冲区看似被丢弃时, 它实际上只是隐藏了起来[2]

[1]译: 还记得吗? 想垂直分隔窗口的话可以用:vertical sfind

[2]译: 规律: hide与vertical都是这样的特殊命令, 它们以一个完整的命令作为参数, 只是以一种方式影响该命令的执行, 同时此类命令并非对所有命令都有效, 它只对那些涉及其影响效果的命令起使用, 如vertical命令只是影响新开窗口的布局, 是水平的还是垂直的.这样它对那些根本不会打开窗口的命令就形同虚设

小心！ 如果你当前有一些被修改内容尚未保存的隐藏缓冲区时，不要草草地退出Vim [1]

非活动缓冲区

一旦一个缓冲区曾被编辑过，Vim就会记下它的一些信息. 这样当它不显示在窗口中并且又不是一个隐藏缓冲区时，它还是会被保留在缓冲区列表中． 这叫非活动缓冲区． 缓冲区的大致类别如下：

List

Active	出现在窗口中，内容被载入
Hidden	不显示在窗口中，但内容被载入
Inactive	不出现在窗口中，内容也未被载入

非活动缓冲区仍被记录在案，因为Vim保留了它的相关信息，比如在其中定义的标记和它的文件名． 这样你可以看到曾经编辑过了哪些文件，也可以再次打开它们.

显示缓冲区列表

下面的命令可以列出整个缓冲区列表：

ex command

```
:buffers
```

命令

ex command

```
:ls
```

与:buffers完全相同，只是看起来没那么顾名思义，它的优点是命令本身很短[2] 结果形如：

Display

```
1 #h   "help.txt"          line 62
2 %l+  "usr_21.txt"        line 1
3      "usr_toc.txt"       line 1
```

第一列是缓冲区编号. 你可以在编辑该文件时以此代替文件名，见下文． 缓冲区编号之后是缓冲区类型标志字符. 然后是文件名和上次退出时光标所在的行号. 可能的缓冲区类型标志字符如下(从左到右)：

[1]译：真要这样退出时Vim还是会提醒你有缓冲区的内容已被改变但尚未保存，除非你声明自己负全责：:qa! 或:wqa

[2]译：(1)ls之于ls就象:grep之于grep，完全不同！(2)Vim中有很多这样的折衷，以较短的命令换取更具描述性的命令名

List

u	未被列出的缓冲区|unlisted-buffer|[[译: 既然未被列出你又怎么知道? :ls!
%	当前缓冲区
#	上一次的活动缓冲区
l	被载入并显示在某窗口中的缓冲区
h	被载入但隐藏的缓冲区.
=	只读的缓冲区
-	不可编辑的缓冲区, 其中'modifiable'选项被关闭
+	有改动的缓冲区

编辑一个缓冲区

你可以以缓冲区编号指定要编辑的缓冲区. 这可以免于键入其文件名:

ex command
```
:buffer 2
```

但是要知道缓冲区编辑的唯一办法就是查找缓冲区列表. 这本身需要执行另一个命令. 你可以用文件名或部分文件名[1]:

ex command
```
:buffer help
```

Vim会根据键入的部分文件名选择一个最为相近的缓冲区. 如果只有一个缓冲区符合条件, 就那直接使用该缓冲区, 本例中是"help.txt".

要在一个新窗口中打开一个缓冲区使用命令:

ex command
```
:sbuffer 3
```

当然这一命令也可以使用文件名.

使用缓冲区列表

你可以用下面的命令来遍历整个缓冲区列表:

List

:bnext	跳转到下一个缓冲区
:bprevious	跳转到前一个缓冲区
:bfirst	跳转到第一个缓冲区
:blast	跳转到最后一个缓冲区

[1]译: 可以看作Vim自动进行了命令补齐, 此时的命令补齐并不要求已经键入的文件名一定是最终文件名的开头字符, 如对于文件filename.txt, 你可以键name然后使用命令补齐, 但同样也可以不用命令补齐, Vim根据键入的部分文件名选择一个最相近的缓冲区

要把一个缓冲区从列表中去除，可以用命令：

ex command
```
:bdelete 3
```

同样，可以使用文件名[1].

如果被删除的缓冲区是活动缓冲区(也就是说被显示在另一窗口)，它所在的窗口就会被关闭. 如果你删除的是当前缓冲区，它所在的窗口也会被关闭. 如果它是最后一个窗口. Vim就会另找一个缓冲区显示在该窗口中. 总不至于让你因此什么都没得编辑.

> 备注: 即使是用":bdelete"命令删除了一个缓冲区Vim还是会记住它. 实际上它被打入"unlisted"列表中，不再显示在":buffers"命令显示的缓冲区列表中. 但":buffers!"还是会让它再度现身(是的，Vim可以完成Misson Impossible). 要彻底清除一个缓冲区，要使用":bwipe". 同时请参考'buflisted'选项.

下一章： |usr_23.txt| 编辑其它文件
版 权： 请参考|manual-copyright| vim:tw=78:ts=8:ft=help:norl:

[1]译：或部分文件名

usr_23.txt Vim 6.3版.最后修改: 2001年09月03日

VIM用户手册--- 作者: Bram Moolenaar

编辑非文本文件

本章内容是关于在Vim中编辑简单的文本文件之外文件. 你可以在Vim中编辑压缩文件或加密文件. 有些文件还需要从网上获取. 除了某些方面的限制之外, 也可以在Vim中象编辑其它文件一样编辑二进制文件.

下一章: |usr_24.txt| 快速键入
前一章: |usr_22.txt| 查找文件
目 录: |usr_toc.txt|

23.1 DOS, Mac 和Unix格式的文件

回想计算机的史前史, 那时的打字机使用两个字符来开始一个新行. 首先是一个字符命令使打印头移回开始位置(回车, <CR>), 然后另一个字符命令控制向前进纸一行(进纸, <LF>).

在计算机诞生之初, 存储设备十分昂贵. 于是有人就提出没有必要用两个字符来表示一行的结束. UNIX一族决定只用进纸一个字符<Line Feed>来表示行尾. 来自苹果阵营的人则把回车<CR>作为换行的标准. MS-DOS(和微软的Windows)仍然决定沿用古老的回车换行<CR><LF>传统.

这也意味着如果把文件从一个系统移到另一个不同的系统, 你就会遇到与换行相关的问题. Vim编辑器则可以识别这些不同格式的文件.

你可以在'fileformats'选项里指定你希望Vim能自动识别的格式的集合. 下面的这个命令就可以让Vim能自动识别UNIX格式和MS-DOS格式:

ex command
```
:set fileformats=unix,dos
```

你在编辑文件时可能就会注意到状态行中关于文件格式的信息. 如果你是在编辑跟本机文件格式相同的文件, 那就不会显示特别的信息, 比如

在Unix系统上编辑一个Unix格式的文件，那没什么好说的． 但如果你编辑一个DOS文件，Vim就会在状态行以下面的形式通知你：

Display
```
"/tmp/test" [dos] 3L, 71C
```

对苹果机的文件格式你会看到"[mac]"的字样．

被检测到的文件格式保存在'fileformat'选项里． 可以用下面的命令查看当前的文件格式：

ex command
```
:set fileformat?
```

[1] Vim以下面的名字代表三种不同的格式：

List

unix	<LF>
dos	<CR><LF>
mac	<CR>

使用苹果机格式

在Unix上，<LF>被用于断行． 通常情况下很少有一行中有一个<CR>字符．但有些情况下，在Vi(和Vim)脚本里会需要该字符作为文本的内容．

但是在Macintosh上，<CR>是一个换行符，<LF>反而可以作为文本的内容出现．

结果就是一个文件同时包含<CR>和<LF>时很难100% 准确地判断是Unix格式还是Mac格式．所以Vim假设你在Unix系统上一般很少去编缉一个Mac格式的文件，对此类格式也就不做检查． 如果你偏要这种格式，可以把"mac" 加到'fileformats'选项中：

ex command
```
:set fileformats+=mac
```

这样Vim会对文件格式作出猜测． 留心看看它什么时候会猜错．

强制指定文件格式

如果你用老版本的Vi去编辑一个MS-DOS格式的文件，你会发现每行的行尾都有一个怪怪的^M字符．（^M其实是回车）．有了自动格式检测就不会这样了． 如果你就是要编辑这样的文件，Vim也允许你强制指定文件格式：

[1]译：fileformat选项并不是一个只读的选项，你可以设置它的值来改变Vim对文件格式的假设

ex command
```
:edit ++ff=unix file.txt
```

"++"字符串告诉Vim后面紧接着的是一个选项名，对该选项的设置将覆盖它的默认值. "++ff"代表的选项是'fileformat'. 你也可以指定为"++ff=mac"或"++ff=dos".

不过并不是每个选项都有这种用法，目前来说只有"++ff"和"++enc"可以这样用.当然也可以用这两个选项的全称"++fileformat"和"++encoding".

格式转换

你也可以利用'fileformat'选项来转换文件的格式. 假如你有一个MS-DOS格式的文件README.TXT. 现在你想把它转换为UNIX格式:

shell command
```
vim README.TXT
```

Vim会识别出这是一个dos格式的文件. 现在把它变为UNIX格式的:

ex command
```
:set fileformat=unix
:write
```

该文件将以Unix格式保存.

23.2 来自因特网的文件

有时候你的email里会有下面这种指定其URL的文件:

Display
```
You can find the information here:
ftp://ftp.vim.org/pub/vim/README
```

当然，你可以用另一个程序把这个文件下载到你本地磁盘上，然后再用Vim打开它.

还有一种更简单的办法. 将光标置于该URL上，使用这个命令:

normal mode command
```
gf
```

运气好的话，Vim会找到一个合适的程序把该文件下载下来并且开始编辑它的一个副本. 要在一个新窗口中打开该文件的话可以用CTRL-W f.

如果中间出了岔子的话你会收到一个错误消息. 可能URL是错误的, 或者你对该文件没有读权限, 网络连接断掉了, 等等. 不幸的是很难说错误的原因是什么. 这时你应该去手工下载该文件.

在Vim中直接访问来自因特网的文件靠的是netrw这个插件. 目前为止可以处理下面几种类型的URL:

List
```
ftp://          uses ftp
rcp://          uses rcp
scp://          uses scp
http://         uses wget (reading only)
```

Vim本身并不处理网络连接, 它有赖于你的系统里相应的程序. 在多数Unix系统上"ftp"和"rcp"程序一般都是默认的配备. "scp"和"wget"可能就需要另外安装.

Vim会在需要开始编辑新文件时检测这些URL, 也包括":edit"和":split". 除了http://之外, 保存命令也可以用.

关于更多的信息比如用户名/密码, 请参考|netrw|.

23.3 加密文件

有些信息你可能想对其它人保密. 比如你要在一台跟学生们共用的电脑里写一个测验, 有些机灵鬼可能会在测验没开始时就把试题搞到手. Vim可以为你的文件进行加密, 这样可以给你一些保护.

要为新编辑的文件加密, 可以在启动Vim时使用"-x"参数, 如:

shell command
```
vim -x exam.txt
```

Vim会向你要一个密码用于加密/解密该文件:

Display
```
Enter encryption key:
```

现在要小心地键入你的密码了. 键入的同时你看不到这些字符, 它们都以星号显示. 为了避免你的键入有误, Vim会要求你再次输入:

Display
```
Enter same key again:
```

现在你可以放心地在文件里写下你的密秘了. 编辑完毕要退出时, 文件在加密后存盘退出.

下次以Vim打开该文件时，它会提醒你输入密码. 这时不需要再用"-x"参数. 你也可以在中用":edit"命令. Vim往加密文件中加入了一个魔术字并据此识别这是一个Vim加密文件[1]

如果你试着用另一程序来打开该文件的话，你会看到一堆乱码. 同样你用Vim编辑但密码不对的话也是乱码一堆. Vim也没办法判断你给的密码是对是错(这也使得破解密码十分困难).

打开或关闭文件加密

要停止对一个文件加密，可以把'key'选项设置为一个空字串：

ex command
```
:set key=
```

下次你存盘该文件时就不会进行加密了.

通过设置'key'的值来进行加密可不是一个好主意，因为密码会显露无遗. 任何趴在你肩膀上的人都能看到你的密码.

为避免这个问题我们创造了":X"命令. 它会象"-x"一样问你要一个密码：

Display
```
:X
Enter encryption key: ******
Enter same key again: ******
```

加密的限制

Vim中所用的加密算法还不够强大. 偶尔防范一下窥视者还可以，对付一个加密专家尤其是他有充足的时间就不行了. 同时你应该知道交换文件并没有被加密，所以在你编辑时拥有超级用户特权的人还是可以从该文件中获取未加密的内容.

有一个避免别人读取你的交换文件的办法就是禁用交换文件. 如果在命令行上指定了-n参数，就不会生成交换文件(Vim会把所有东西都放到内存里). 比如，下面的命令就在编辑加密文件"file.txt"时不使用交换文件：

shell command
```
vim -x -n file.txt
```

[1]译：魔术字是应用程序对特定文件格式的一种约定. 但就象文件扩展名一样，它并非强制性的，一个文件可以以某个类型的文件的魔术字作为伪装，但应用程序会在处理文件的其它部分时检测出错误，魔术字一般都在文件的固定位置，多位于文件最开头的几个字节，如MS-DOS可执行文件以MZ作为其魔术字. Tiff文件以0x4949作为其魔术字

如果已经在编辑过程中，也可以通过下面的命令禁用交换文件：

ex command
```
:setlocal noswapfile
```

因为没有了交换文件，所以灾难恢复也不可能了． 这时最好是经常保存编辑的结果，免得一番辛苦的成果杳然无踪．

当文件在内存中时，它是以普通文本的形式保存的． 任何有足够权限的人还是可以查看编辑器的内存和文件的内容．

如果你还用到了viminfo文件，要注意文本寄存器也可能会把你的机密在这里泄露．

如果你的确是要高度保密你的文件，最好在一个没有联网的电脑上编辑，使用足够强大的加密工具，用完就把电脑锁在一个安全的地方．

23.4 二进制文件

你也可以用Vim来编辑二进制文件．不过Vim并未打算对二进制文件提供支持． 所以还是有一些限制．但是至少你可以用它读取一个文件，改变一个字符并把它存盘写回去，结果是只有被编辑的字符内容变了，其它部分保持原来的内容．

要保证Vim在编辑二进制文件时没有滥用它惯常的聪明办法，你需要在启动Vim时使用"-b"参数：

shell command
```
vim -b datafile
```

这会设置'binary'选项．设置该选项的作用是避免你不希望有的副作用． 比如'textwidth'会被设置为0，禁止了自动换行． 文件总是以Unix格式读入．

用Vim的二进制模式进行编辑可以用来修改一个可执行程序中的文本信息．不过注意此时只是去修改，不要去插入或删除任何东西，这样做会让你的程序死得很难看． 用"R"进入替换模式进行修改倒是个不错的主意．

二进制文件中很多字符都是不可打印字符．设置下面的选项可以以十六进制格式显示这些字符：

ex command
```
:set display=uhex
```

另外，"ga"命令可以来查明当前光标下字符的本来面目． 以<Esc>[1]字符为例，其输出格式是：

[1]译：最左上角的字符

Display
```
<^[>  27,  Hex 1b,  Octal 033
```

二进制文件里也可能出现超长的行. 如果仅是想看个大概就可以把'wrap'选项关闭:

ex command
```
:set nowrap
```

字节位置

下面的命令可以让你获知光标所在字符是整个文件中第几个字节:

normal mode command
```
g CTRL-G
```

输出信息比ga命令更为丰富:

Display
```
Col 9-16 of 9-16; Line 277 of 330; Word 1806 of 2058; Byte 10580 of 12206
```

最后出现的两个数字分别是当前字符在整个文件中是第几个字节[1]及全部的字节数. 这个统计会把因'fileformat'选项而起的字节数变量也计算在内.

"go"命令可以移动到文件中指定字节去, 比如下面的命令就可以转到第2345字节的位置去:

ex command
```
2345go
```

使用xxd程序

一个地道的二进制文件编辑器会以两种方式显示内容: 通常的文本显示方式和十六进制格式. 在Vim中要收到这种效果你可以先用"xxd"程序来做转换. 该程序随Vim一起发布.

首先还是以二进制方式开始编辑:

shell command
```
vim -b datafile
```

现在用xxd程序把文件进行十六进制格式的转储:

[1]译: 以1开始计数, 即第一个字节显示1而不是0

ex command

```
:%!xxd
```

结果形如：

Display

```
0000000: 1f8b 0808 39d7 173b 0203 7474 002b 4e49  ....9..;..tt.+NI
0000010: 4b2c 8660 eb9c ecac c462 eb94 345e 2e30  K,.`.....b..4^.0
0000020: 373b 2731 0b22 0ca6 c1a2 d669 1035 39d9  7;'1.".....i.59.
```

现在你可以随心所欲地浏览和编辑了，Vim将之视为普通文本． 改变其十六进制不会引起右边对应字符的改变，反之也一样[1]．

编辑完成后再做一次逆向转换：

ex command

```
:%!xxd -r
```

逆向转换时只有其十六进制形式被认为是有效的． 对可打印形式的改变会被转换程序忽略．

请参考xxd的参考手册获取更多关于该程序使用的信息．

23.5 压缩文件

其实简单： 在Vim中你可以象编辑其它文件一样直接编辑一个压缩文件．"gzip"插件会在你编辑时处理解压的问题．保存时进行压缩．

目前支持的压缩方法有：

List

```
.Z      compress
.gz     gzip
.bz2    bzip2
```

Vim使用上面这些程序进行压缩和解压．如果系统中还没有的话你需要在使用这一功能前先安装这些程序．

下一章： |usr_24.txt| Inserting quickly
版　权： 请参考|manual-copyright| vim:tw=78:ts=8:ft=help:norl:

[1]译：一个真正的二进制文件编辑器如UltraEdit或WinHex会做这种同步，同时，只有对其十六进制形式做出改变时用xxd -r才能真正改变文件

usr_24.txt Vim 6.3版.最后修改: 2003年08月18日

VIM用户手册--- 作者: Bram Moolenaar

快速键入

对于键入文字的工作, Vim也提供了一些方法来减少击键, 避免错误. 插入模式下的补齐功能可以重复输入已经录入的word. 以较短的word缩写来代替键入整个长的word. 也可以录入你键盘上本不存在对应键的字符.

下一章: |usr_25.txt| 编辑格式化的文本
前一章: |usr_23.txt| 编辑其它文件
目　录: |usr_toc.txt|

24.1 纠错

<BS>键已经提到过, 它删除光标之前的字符. <Del>删除光标所在处(光标之后)的字符.

如果你发现整个word键入有误, 可以使用CTRL-W:

Display
```
The horse had fallen to the sky
                               CTRL-W
The horse had fallen to the
```

如果发现整行键入的内容都弄乱了, 可以使用CTRL-U删除它来重新开始. 同时这会保留光标之后的字符并且保持原有的缩进. 只有第一个非空白字符之后的内容才会被删除. 在下例中光标位于"fallen"的"f"上时按下CTRL-U:

Display
```
The horse had fallen to the
              CTRL-U
fallen to the
```

继续键入几个word之后才发现前面某个word有误时，就要靠移动光标进行定位，然后更正它． 比如你键入了以下内容：

Display
```
The horse had follen to the ground
```

如果要把"follen"改为"fallen"． 而当前光标在行尾，你需要用以下命令：

List

退出Insert模式	<Esc>
后退4个word	4b
将光标移到"o"上	l
把"o"改为"a"	ra
回到Insert模式	A

另一个办法是：

List

后退4个word	<C-Left><C-Left><C-Left><C-Left>
移到"o"上	<Right>
删除"o"	<Del>
插入一个"a"	a
将光标移到行尾	<End>

这里用的是特殊来进行光标移动，当前的Insert模式保持不变．与一个一般的无模式编辑器没有两样． 也很好记，不过它更费时间(你需要不断地把手在字母键区和光标键之间移动，另外，要不瞄一眼的话你也很难一次按准<End>键)．特殊键用于定义一个不离开Insert模式进行操作的映射时很有用． 可以避免你多键入额外的命令． 下面是一个在Insert模式下可用特殊键的小计：

List

<C-Home>	到文件头
<PageUp>	向上滚屏
<Home>	到行首
<S-Left>	向左移动一个word
<C-Left>	向左移动一个word
<S-Right>	向右移动一个word
<C-Right>	向右移动一个word
<End>	到行尾
<PageDown>	向下滚屏
<C-End>	到文件尾

除此之外在|ins-special-specified|还列出了一些特殊键.

24.2 显示匹配字符

键入)字符时要是能看出来它与前面的哪个(字符匹配就太好了. 要使Vim具有此功能只需:

ex command
```
:set showmatch
```

现在你再键入"(example)", 一旦按下了)字符Vim就会把光标移到前面的(字符上, 停留半秒钟, 然后回到)字符之后的位置. 如果没有找到相匹配的(字符, Vim会以蜂鸣警告. 这样你就可以知道可能在哪里漏了一个(字符, 或者多输入了一个)字符. 对[]和{}这样成对的括号也与()一样. 不需要等到键入闭括号之后的下一个字符, 一旦你键入了某个闭括号,ZZ 等待时间的长短可以由选项'matchtime'来控制. 比如说让Vim等待一秒半钟:

ex command
```
:set matchtime=15
```

时间单位是十分之一秒.

24.3 自动补全

Vim可在编辑时自动补全一个词[1]. 首先键入一个词前面的部分[2], 然后按下CTRL-P, Vim会据此补全整个词.

假如你以编辑一个C程序希望缩写如下一个语句:

code
```
total = ch_array[0] + ch_array[1] + ch_array[2];
```

已经键入了以下部分:

code
```
total = ch_array[0] + ch_
```

此时, 你可以使用CTRL-P告诉Vim去补全ch_的其余部分. Vim会搜索以此开头的所有word, 此例中它查找以"ch_"开头的word, 结果是ch_array. 所以CTRL-P会补全这个变量名的剩余部分:

---

[1]译注: 对于中文用户是补全一个句子, Vim尚无法做到识别中文意义上的词
[2]译注: 前面的部分到底是几个字母, 往下看

```
code
total = ch_array[0] + ch_array
```

再键入一些内容后当前行变成这样(以空格结束):

```
code
total = ch_array[0] + ch_array[1] +
```

此时再按下CTRL-P的话Vim还是搜索以光标前的第一个word开头的word. 但这里光标前是空格，所以它会往前查找第一个word，"ch_array". 再按下CTRL-P会继续下一个word: "total". 第三次按CTRL-P会继续往回找，如果实在没东西可找了，它就返回你原来输入的那一部分的word，此例中是空. 第4次按下CTRL-P又会循环往前查找，又是"ch_array".

使用CTRL-N可以往前查找word来补全. 因为查找达到文件头尾时回绕过去循环进行，所以CTRL-N和CTRL-P往往会找到相同的word来补全，只不过查找的顺序相反. 提示:CTRL-N意为Next-match[1]，CTLR-P意为Previous-match[2].

Vim会尝试多种办法来补全一个word. 默认情况下，它会查找以下这些地方:

1. 当前文件
2. 在其它窗口打开的文件
3. 其它载入缓冲区的文件(隐藏的缓冲区)
4. 没有载入的文件(非活动的缓冲区)
5. Tag文件
6. 当前文件中所有被#include语句引入的头文件

选项

可以用'complete'选项来定制Vim在补全word时所用的策略.

查找时会隐含地使用'ignorecase'选项. 设置了该选项时，会在搜索匹配的word时忽略大小写的不同.

对于自动补全有一个选项十分有用，它就是'infercase'. 它使搜索匹配的word时忽略剩余部分的大小写(当然还得'ignorecase'被设置了才行)，但继续保留已键入的部分的大小写. 这样对于键入了"For"时Vim会查找到"fortunately"这样的匹配，但最终的结果是"Fortunately".

[1]译注: 下一个匹配
[2]译注: 前一个匹配

补全特殊的文档元素

如果你自己清楚要找的东西，你可以用以下命令来补全这样一些特殊的文档元素：

List
```
CTRL-X CTRL-F         文件名
CTRL-X CTRL-L         整行内容
CTRL-X CTRL-D         宏定义(也包括那些在include文件里定义的宏)
CTRL-X CTRL-I         当前文件和被当前文件include的文件
CTRL-X CTRL-K         来自一个字典文件的word
CTRL-X CTRL-T         来自一个thesaurus的word
CTRL-X CTRL-]         tags
CTRL-X CTRL-V         Vim的命令行
```

键入这些特殊命令后再使用CTRL-N可以往下查找符合的匹配，CTRL-P则往下查找.

关于这些命令的更多信息请参考：|ins-completion|.

补全文件名

我们以CTRL-X CTRL-F来作为一个例子. 这个命令查找文件名. 它会查找当前目录下的文件，看哪些文件名以你眼下键入的word开头. 比如说你的当前目录下有这几个文件：

List
```
main.c  sub_count.c  sub_done.c  sub_exit.c
```

现在在插入模式下你键入了：

Display
```
The exit code is in the file sub
```

此时，按下CTRL-X CTRL-F命令. Vim会查看当前目录下哪些文件以"sub"开头. 第一个符合条件的是sub_count.c. 如果这不是你要的那个文件名，按CTRL-N选下一个，是sub_done.c. 再按CTRL-N得到的是sub_exit. 结果如下：

Display
```
The exit code is in the file sub_exit.c
```

如果文件名以/开头(Unix)或者是C:\(MS-Windows)的话查找的范围会扩大到对应的文件系统. 比如按下"/u"和CTRL-X CTRL-F. 这会匹配到"/usr"(在Unix上)：

Display
```
the file is found in /usr/
```

现在再按下CTRL-N就会回到"/u". 如果你要的正是"/usr/"并且想继续找它下面的内容，再用一次CTRL-X CTRL-F:

Display
```
the file is found in /usr/X11R6/
```

当然，实际的结果要看你的文件系统中的具体内容. 文件匹配的依据是字母顺序. (译注: 简单说如果键入了"file_" CTRL-X CTRL-F而当前目录下只有"file_a", "file_b"两个文件，那么"file_a"将是第一个被匹配的，然后是"file_b")

24.4 重复录入

如果按下了CTRL-A，编辑器会插入你上一次在insert模式下录入的内容.

假如你有一个文件以下面的行开头:

code
```
"file.h"
/* Main program begins */
```

你编辑这个文件并在第一行开头插入了"#include ":

code
```
#include "file.h"
/* Main program begins */
```

然后用"j^"命令到了下一行的开头. 现在要想在此也插入"#include". 使用:

normal mode command
```
i CTRL-A
```

结果将是:

code
```
#include "file.h"
#include /* Main program begins */
```

因为前一次你在insert模式下已经键入过文字"#include "所以这里按CTRL-A会重复插入它. 现在按键入其余部分"main.h"<Enter> 完成该行的编辑:

code
```
#include "file.h"
#include "main.h"
/* Main program begins */
```

CTRL-@命令基本与CTRL-A一样，不同是它在插入之后会退出Insert模式. 如果实际的编辑任务就是简单地插入上一次录入过的内容的话，它倒是比CTRL-A省事一些.

24.5 从其它行复制

CTRL-Y命令会插入当前光标之上的一行中相同位置字符. 如果你要复制上一行中的内容,这一命令就十分有用了，例如，你有如下的C代码:

code
```
b_array[i]->s_next = a_array[i]->s_next;
```

现在你要键入同样的一行内容，只不过要把"s_next"替换为"s_prev". 开始新行的内容，按14次CTRL-Y，一直到"next"中"n"的位置:

code
```
b_array[i]->s_next = a_array[i]->s_next;
b_array[i]->s_
```

现在键入"prev":

code
```
b_array[i]->s_next = a_array[i]->s_next;
b_array[i]->s_prev
```

继续按CTRL-Y重复上一行中同列的字符直到下一个"next":

code
```
b_array[i]->s_next = a_array[i]->s_next;
b_array[i]->s_prev = a_array[i]->s_
```

现在再键入"prev;"完成整行内容.

CTRL-E与CTRL-Y十分相似，不过它插入的是当前行之下的那一行中同列的字符[1].

24.6 插入一个寄存器的内容

[1]译注: 提示: 想一想CTRL-E，CTRL-Y在Normal模式下的功能

命令CTRL-R {register}可以在当前位置插入指定寄存器的内容. 这可以避免手工键入一个过长的word. 比如要键入以下行:

code
```
r = VeryLongFunction(a) + VeryLongFunction(b) + VeryLongFunction(c)
```

假设函数是在另一个文件中定义的. 打开那个文件并把光标置于该函数名上[1], 把它复制到寄存器v 中:

normal mode command
```
"vyiw
```

其中"v是指定寄存器的特殊记法, "yiw"命令是复制一个word本身. 现在回到刚才的文件, 键入该行开头的几个字母:

Display
```
r =
```

现在可以使用CTRL-R v来插入函数的名字:

code
```
r = VeryLongFunction
```

接下来就是继续键入非函数名的字符, 然后仍用CTRL-R v插入函数名. 你可能已经想到补全功能可以做同样的事, 不过使用寄存器有另一个优点就是如果有很多word都以相同的字符开头时它会比补全功能快, 因为它不需要连续按CTRL-N或CTRL-P来遍历到你要的word.

如果寄存器的内容中包含了象<BS>这样的特殊字符, 它们的功能就会象直接从键盘键入一样[2]. 如果你真正想要做的就是插入这样一个特殊字符, 用CTRL-R CTRL-R {register}.

24.7 缩写

缩写简单说就是以短代长. 比如"ad"代表"advertisement". Vim可以让你输入短的缩写然后自动扩展为长的全名.

下面的命令告诉Vim你想在每次键入"ad"时都自动扩展为"advertisement":

ex command
```
:iabbrev ad advertisement
```

[1]译注: 注意并不必需把光标置于该函数名的开头
[2]译注: 也就是说, <BS>就是删除前面的一个字符

现在，每次你键入"ad"，完整的word "advertisement"就会被插入到当前位置. Vim根据你键入一个非word字符来判断进行缩写替换的时机，比如一个空格:

List

已经键入的部分	键入后你会看到的部分
I saw the a	I saw the a
I saw the ad	I saw the ad
I saw the ad<Space>	I saw the advertisement<Space>

仅仅键入了"ad"可并不会发生缩写替换. 这样你才可以输入象"add"这样的word，进行缩写替换时只检查一整个的word是否符合一个缩写的定义.

缩写--育繁于简

可以定义这样的缩写: 它可以扩展为多个word，比如下面的命令可以定义"JB"扩展为"Jack Benny":

ex command
```
:iabbrev JB Jack Benny
```

作为一个程序员，我经常使用下面两个不太常见的缩写:

ex command
```
:iabbrev #b /****************************************
:iabbrev #e <Space>****************************************/
```

这两个缩写用于创建一个看起来象矩形文字块的注释. 注释以#b开头，构画出第一行. 接下来键入注释的内容，最后用#e完成底行. 注意#e缩写要扩展的内容开头处有一个空格. 或者说，开头的两个字符是空格-星号. 通常情况下Vim会忽略缩写和它的扩展全名之间的空格. 这里我键入7个字符<, S, p, a, c, e, >就是为了避免空格被忽略掉.

> 备注: 可以用":iab"代替完整的命令名":iabbrev"，瞧，缩写定义本身就应用了缩写!

更正打字错误

有些词被打字员拼写错误的频繁很高，比如把"the"拼成"teh". 缩写的一个副作用是可以更正这些错误，看下面:

ex command
```
:abbreviate teh the
```

你完全可以定义一个错误率较高的词的一个列表，每行一个，专用于更正这些拼错的词.

列出已定义的缩写

":abbreviations"命令可以列出当前定义的所有缩写:

Display

```
:abbreviate
i  #e           ****************************************/
i  #b          /****************************************
i  JB          Jack Benny
i  ad          advertisement
!  teh         the
```

第一列中的"i"表明这是一个用于Insert模式下的缩写. 这样的缩写只在Insert模式下有效. 此外还可以有下面的字符代表该缩写发生作用的工作模式:

List

c	命令行模式	:cabbrev
!	同时适用于Insert模式和命令行模式	:abbreviate

命令行模式下的缩写用处并不大，用的最多的还是":iabbrev"命令. 这也避免了象下面的命令中"ad"被替换:

ex command

```
:edit ad
```

删除缩写

":unabbreviate"命令可以用于删除一个缩写. 假设你已定义了下面的缩写:

ex command

```
:abbreviate @f fresh
```

就可以用命令

ex command

```
:unabbreviate @f
```

来去除它. 当你键入这个命令时，你会发现在该命令中@f还是被替换为了"fresh"[1]. 不要管它，Vim知道你的意思(除非"fresh"本身又被定义为了一个缩写，不过这种事也太少见了). 还有一个命令可以移除所有的缩写:

[1]译注: 这是它临死前的最后一次替换

ex command
```
:abclear
```

":unabbreviate"和":abclear"也有几种变体专用于Insert模式(":iunabbeviate"和":ia
命令行模式(":cunabbreviate"和":cabclear").

嵌套缩写

定义一个缩写时要注意一件事： 作为缩写替换结果的字串不应该再被某个缩写扩展. 比如:

ex command
```
:abbreviate @a adder
:imap dd disk-door
```

你一键入@a就会被替换为"adisk-doorer"[1]. 你不是想要这个吧. ":noreabbrev"命令可以避免在定义缩写时再被其它的缩写所扩展:

ex command
```
:noreabbrev @a adder
```

不过话说回来，缩写结果里本身又碰巧包含了另一个缩写的情况毕竟少见.

24.8 键入特殊字符

CTRL-V命令可以保证你键入的下一个字符被原封不动地被录入. 也就是说，该字符所具有的任何特殊意义都被忽略. 比如:

normal mode command
```
CTRL-V <Esc>
```

这会插入一个escape字符. 而不是让你离开Insert模式. (不要在CTRl-V后面加空格，上面示例中的空格只是为了提高可读性)[2].

> 备注: 在MS-Windows上CTRL-V是粘贴命令的快捷键. 此时应用CTRL-Q来替代它. 对Unix来说，CTRL-Q 又不能在某些终端上正常工作，因为它也有特殊的意思[a].
>
> [a]译注: 什么特殊意思

[1]译注: 而且这里的dd 后面也不需要跟一个非word字符才会被扩展，它在一个word内部也可马上被扩展

[2]译注: 从技术上来讲，escape字符只是一个ASCII为27的字符而已，只不过它被多数应用软件赋予了"撤消"或"退出"这样的功能含义

你也可以使用CTRL-V {digits}来插入一个由{digits}指定其ASCII码的字符. 比如, ASCII为127的字符是<Del>(但不需要按下<Del>键!). 插入<Del>字符可以:

normal mode command
```
CTRL-V 127
```

用这种方法你可以插入0到255的所有字符.如果你键入的数字少于两个, 那么Vim会在遇到一个非数字字符时终止这个命令. 要避免非得键入一个非数字字符才能让这个命令结束,你可以在数字前加上一个或两个0来凑足3个数.

现在的3个命令都会插入一个<Tab>和一个点号:

Display
```
CTRL-V 9.
CTRL-V 09.
CTRL-V 009.
```

要用十六进制来表示你的ASCII, 在CTRL-V后面附加一个"x":

normal mode command
```
CTRL-V x7f
```

同样可以键入所有的256个字符(ASCII为255的字符用CTRL-V xff). 你也可以用"o"让Vim把接下来的数字视为8进制的,接下来的两个方法还可以让你键入一个16bit或32bit的数字(比如,用来指定一个Unicode字符):

normal mode command
```
CTRL-V o123
CTRL-V u1234
CTRL-V U12345678
```

24.9 连体字

有一些字符在键盘上没有对应的键. 比如表示版权的字符. 要在Vim中输入这些字符可以使用digraphs, 它用两个字符来表示一个有意义的符号. 要输入一个?号, 可以通过连续键入三个键:

normal mode command
```
CTRL-K Co
```

命令

ex command
```
:digraphs
```

可以让你查看都有哪些digraphs可用.

Vim会显示一个digraph的列表. 下面是一个示例性的内容:

Display
```
AC ? 159  NS ? 160  !I ? 161  Ct ? 162  Pd ? 163  Cu ? 164  Ye ? 165
BB ? 166  SE ? 167  ': ? 168  Co ? 169  -a ? 170  << ? 171  NO ? 172
-- ? 173  Rg ? 174  'm ? 175  DG ? 176  +- ? 177  2S ? 178  3S ? 179
```

举例来说, 上面的示例内容的意思是如果你键入CTRL-K Pd结果是输入了字符(?. 该字符ASCII为163(十进制).

Pd代表Pound. 大多数的Digraphs都有一个颇具提示意味的缩写名. 看一下它的列表内容你就是明白其中的规律.

你可以交换Digraphs的第一个和第二个字符, 如果交换后的这个二字符的组合并没有碰巧是另一个Digraphs时, 结果将是一样的, CTRL-K dP就跟CTRL-K Pd一样. 因为"dP"并不是Vim已定义的一个Digraphs.

> 备注: Digraphs的工作依赖于Vim所使用的字符集. MS-DOS就与MS-Windows不同. 最好经常用":digraphs"看看到底有哪些可用的Digraphs.

你也可以定义你自己的digraphs. 比如:

ex command
```
:digraph a" ¦d
```

这个定义是说CTRL-K a"会实际插入一个?字符. 你也可以用一个十进制数来代表要插入的字符. 下面的命令效果是一样的:

ex command
```
:digraph a" 228
```

关于Digraphs的更多内容请参考: |digraphs|

插入特殊的另一种办法是使用键映射. 参考|45.5|了解详细信息.

24.10 Normal模式命令

Insert模式所提供的命令功能是十分有限的. 在Normal模式下可就丰富多了. 通常情况下你要用Normal模式下的一个命令时都会用<Esc>来退出Insert模式, 执行完Normal模式下的命令时再用"i"或"a"重新进入Insert模式.

针对这种情形Vim提供了一个快速的办法. 使用CTRL-O {command}你可以在Insert模式下执行任何一个Normal模式下的命令. 比如, 要删除从当前光标到行尾的字符:

normal mode command
```
CTRL-O D
```

这种快捷办法只允许你一次执行一个Normal模式的命令. 但是你可以为这个命令指定一个寄存器名或命令计数. 下面是一个复杂一点的例子:

normal mode command
```
CTRL-O "g3dw
```

这个命令会删除3个单词, 并在寄存器g中记下它们.

下一章: |usr_25.txt| 编辑格式化的文本
版 权: 请参考|manual-copyright| vim:tw=78:ts=8:ft=help:norl:

usr_25.txt Vim 6.3版.最后修改：2003年06月21日

VIM用户手册--- 作者： Bram Moolenaar

编辑格式化的文本

文本文件很少是一行一个句子这样规整． 本章讲述如何把一个句子分段来合乎页面的外观要求以及其它有关格式化的东西．Vim也同样有一些特性专用于编辑单行成段的文本和行列分明的表格数据．

下一章： |usr_26.txt| 重复重复，再重复
前一章： |usr_24.txt| 快速键入
目 录： |usr_toc.txt|

25.1 断行

Vim有一些功能大大便利了文本的处理． 默认情况下，编辑器并不会自动换行． 也就是说你要手工按下回车才可以换行． 这在写程序时是需要的，因为你要自己决定一个语句应在何处断行． 但如果你在写文档，希望每行最多有70个字符时，这可不是一件美差．

如果你设置了'textwidth'选项，Vim就会自动换行． 假设你想限制一行最多有30个字符．可以使用下面的设置：

ex command
```
:set textwidth=30
```

现在再键入下面的内容(特意增加了标尺)：

Display
```
         1         2         3
123456789012345678901234567890123456
I taught programming for a whi
```

如果你接下来按了"l"字符，该行的长度就已经超过30个字符了.这时Vim就会在此处断行：

Display
```
          1         2         3
12345678901234567890123456789012345
I taught programming for a
whil
```

继续编辑，键入该段的其余文本：

Display
```
          1         2         3
12345678901234567890123456789012345
I taught programming for a
while. One time, I was stopped
by the Fort Worth police,
because my homework was too
hard. True story.
```

你不需要敲回车键，Vim会自动为你断行.

备注：'wrap'选项使Vim能显示需要折行的过长的行，但这是另一回事，它只是为了显示的需要，并不会在文件中实际插入一个换行符.

重新格式化

Vim编辑器不是一个字处理器． 在一个字处理器中，你若是在一段的开头删除一些东西，换行的处理会被重新考虑．在Vim中不行；所以你若是删除了上例中第一行的"programming" 这个单词，结果就只是这一行变短了而已：

Display
```
          1         2         3
12345678901234567890123456789012345
I taught for a
while. One time, I was stopped
by the Fort Worth police,
because my homework was too
hard. True story.
```

这可不妙． 要让文本段恢复形象你可以使用"gq"操作符命令． 我们先来看Visual选择区，在上例中的第一行键入以下命令开始：

normal mode command
```
v4jgq
```

"v"命令进入Visual模式，"4j"移动到段尾然后执行"gq"命令，结果将是：

Display
```
         1         2         3
12345678901234567890123456789012345
I taught for a while. One
time, I was stopped by the
Fort Worth police, because my
homework was too hard. True
story.
```

备注：还可以按特殊的文本布局进行格式化，请参考|auto-format|.

因为"gq"是一个操作符命令，所以你可以使用此类命令所支持的3种办法来指定它的作用对象：Visual模式，使用移动光标的命令和文本对象.

上面的示例也可以用"gq4j"命令． 区别是可以少敲几个字符，但是你需要计算行数.一个更有用的移动命令是"}"． 它会移动到一段的末尾．所以"gq}"会格式化当前光标至当前段尾的文本.

一个使用"gq"时非常常用的文本对象是段． 试一下命令：

normal mode command
```
gqap
```

"ap"意为"a-paragraph"． 这个命令会格式化一段文本(段以空行为分界)． 包括当前光标之前的部分.

如果你的段已经是以空行分隔，你可以用下面的命令格式化整个文件：

normal mode command
```
gggqG
```

"gg"移动到第一行，"gqG"将格式化进行到底(行).

警告： 如果你的段并不符合上述标准，就会在格式化时被连为一段进行处理． 通常的错误是以一个含有空格或制表符的行来分隔所谓的"段"． 这是空白行，不是空行.

Vim不光可以格式化普通文本文件．查看|fo-table|了解如何处理这些非常规的文件．'joinspaces'选项可以用来控制句子之间的空白间距.

也可以使用一个外部的程序来进行格式化． 这在你的文件不能以Vim的内置命令很好地格式化时尤其有用． 参看'formatprg'选项.

25.2 对齐文本

要让一个范围的行居中，使用下面的命令：

ex command
```
:{range}center [width]
```

{range}是一个通常的命令行范围．[width]是一个用于指定行宽的可选参数． 如果不指明[width]，它的默认值取自'textwidth'． （如果'textwidth'的值是0，就取80）． 如：

ex command
```
:1,5center 40
```

结果如下：

Display
```
   I taught for a while. One
  time, I was stopped by the
 Fort Worth police, because my
  homework was too hard. True
            story.
```

右对齐

近似地，":right"命令可以使文本右对齐：

ex command
```
:1,5right 37
```

将使结果是：

Display
```
    I taught for a while. One
   time, I was stopped by the
Fort Worth police, because my
  homework was too hard. True
                       story.
```

左对齐

最后还有：

ex command
```
:{range}left [margin]
```

与":center"和":right"不同，":left"的可选参数的意义不再是行的长度，而是指左边留白的宽度．如果不予指明，就将使每行内容都向显示窗口最左边靠齐(等同于使用0宽度的左边留白)． 如果指定了5，所有文本就都会向右缩进5个空格．如，使用下面的命令：

ex command
```
:1left 5
:2,5left
```

后结果文本将是

Display
```
     I taught for a while. One
time, I was stopped by the
Fort Worth police, because my
homework was too hard. True
story.
```

左右对齐

Vim并没有一个内置的命令来使文本左右对齐． 不过，有一个不错的宏包可以实现它一功能．要使用该宏包，执行下面的命令：

ex command
```
:runtime macros/justify.vim
```

这个脚本定义了一个新的visual模式下的命令"_j"． 要调整一段文本使之左右对齐，只需在Visual模式下选定这段文本，然后执行"_j"．

要了解其来龙去脉还最好还是亲自看一下这个脚本做了什么． 要打开该文件，在光标位于下面的文件名上时按"gf"：

Display
```
$VIMRUNTIME/macros/justify.vim.
```

另一个变通办法是使用一个外部程序．如：

ex command
```
:%!fmt
```

25.3 缩进和制表符

缩进可以使文本突出显示． 本教程中的例子都是以8个空白的字符宽度或一个制表符来进行缩进的[1]． 通常要做的只是在一行开头键入一个制表符:

```
Display
        the first line
        the second line
```

需要键入的东西是制表符，一些文字，回车，再一个制表符，和其后的文字． 'autoindent'选项可以自动插入缩进:

```
ex command
        :set autoindent
```

接下来的新行将会沿用其前一行中所使用的缩进． 在上例中，回车之后就不再需要键入制表符了．

增加缩进

要增加一行的缩进量，使用">"操作符命令． 通常情况下人们喜欢使用">>"命令，这会增加当前行的缩进量．

每次缩进量增减的单位由选项'shiftwidth'指定． 其默认值是8．要让">>"命令只增加4个字符宽度的额外缩进量，这样设置该选项:

```
ex command
        :set shiftwidth=4
```

在上例中的第二行文本上使用命令">>"，结果将是:

```
Display
        the first line
            the second line
```

"4>>"将会增加4行[2]的缩进量．

缩进量

如果你想让缩进量是4的倍数，只需要把'shiftwidth'设为4即可． 但是键入制表符还是会插入一个8字符宽度的缩进． 这可以通过'softtabstop'选项得以改观:

[1]译者: 为使例子不致超出方框，我早已打破了这一风格．不过显眼的框框会让它看起来更加醒目

[2]译注: 包括当前行在内

ex command
```
:set softtabstop=4
```

这将会让一个制表符只插入4个字符宽度的缩进. 如果已经有了4个字符宽度的空白，就会真正插入一个制表符来代替这总的8个字符(省了7个字符). (如果你根本不想用任何制表符，你可以打开'expandtab'选项)

备注: 你也可以把'tabstop'选项设置为4. 不过，如果你在'tabstop'设置为8时打开该文件可能看起来就面目全非了. 被其它程序处理或在打印时也可以会出问题. 所以这里还是建议把'tabstop'设置为8. 毕竟这是大众标准.

调整制表符

如果你在编辑文件时使用的制表符宽度是3. 在Vim里就会很难看，因为一般的制表符宽度都是8. 当然你可以把'tabstop'选项设为3. 但每次你在Vim里打开这个文件都要为它特别一个制表符宽度.

Vim可以改变文件中已有的制表符宽度. 首先，设置'tabstop'的值来调整缩进的外观，然后使用":tab"命令:

ex command
```
:set tabstop=3
:retab 8
```

":retab"命令会把'tabstop'改为8，这样改变后的文件看起来还是一样. 这个命令会把文件中连续的空白替换成制表符或空格. 现在你可以保存这个文件了，下次再打开时就不用额外为它设置制表符宽度了.

警告: 在源程序里使用":retab"时，它也会改变一个字符串常量中的制表符. 所以最好是在字符串常量中使用"\t"来代替实际键入一个制表符.

25.4 处理长的文本行

有时候人们需要编辑的文本行长度会超出屏幕能显示的列宽. 这时Vim会把这一行折叠到下一行去显示.

如果你把'wrap'选项关闭，那不管长行短长就都只会占据屏幕上的一个行. 超出屏幕显示的部分就看不见了.

一旦你把光标移动到这些在屏幕上看不见的字符上，它们就会被移到屏幕上显示，Vim会使该行内容自动向左滚动. 这就好象在水平移动一个视窗一样.

默认情况下，使用GUI的Vim不会显示一个水平滚动条． 如果你想用，就要使用下面的命令：

ex command
```
:set guioptions+=b
```

这时一个水平滚动条就会显示在Vim窗口的底部．

如果没法使用滚动条或者有你也不想用它[1]，你可以用下面的命令来左右移动一个文本行． 文本左右移动时，光标还是保持不动，除非为了显示文本它必需移动．

List

zh	向右滚动
4zh	向右滚动4个字符
zH	向右滚动半个窗口的宽度
ze	向右移动使当前光标成为最右端的可见字符
zl	向左滚动
4zl	向左滚动4个字符
zL	向左滚动半个窗口
zs	向左移动使当前光标成为最左边的可见字符

我们找一个长一点的文本行来试一试．假设下面的例子中光标位于"which"中的"w"字符上． "current window"这个标尺指示的是当前的可视区． 下面带"window"字样的标尺指示执行命令后的可视文本区

Display
```
                          |<-- current window -->|
        some long text, part of which is visible in the window
ze        |<--     window     -->|
zH         |<--     window     -->|
4zh               |<--     window     -->|
zh                   |<--     window     -->|
zl                     |<--     window     -->|
4zl                       |<--     window     -->|
zL                             |<--     window     -->|
zs                            |<--     window     -->|
```

关闭折行显示时的行内移动

'wrap'关闭状态下左右滚动文本行时，可以使用下面的命令在当前可视区移动光标．窗口可视区左右的文本都不受影响．这些命令也绝不会引起左右滚动．

[1]译注： 多数的Vim用户都更喜欢命令行形式而很少使用GUI所提供的功能，包括Bram本人

List
```
g0                到窗口内的第一个字符
g^                到当前窗口内第一个非空白字符
gm                到当前窗口中间的字符上
g$                到当前窗口的最后一个字符上
         |<--     window    -->|
some long    text, part of which is visible
         g0  g^    gm              g$
```

word的绕行显示

edit-no-break

有时候在生成被另一个程序读取的文件时，可能会要求一段内容不能有断行．但使用'nowrap'的话又不能看到整个句子的全貌．'wrap'选项打开的话，又可能会把一个词从中间硬生生折到下一行去显示，让你看起来很费劲．一个好办法是仍然打开'wrap'，同时打开'linebreak'选项．这样Vim就会在适当的地方折叠显示长的文本行．同时文本的内容本身不受影响．

不打开'linebreak'选项时：

Display
```
+----------------------------------+
|letter generation program for a b|
|ank.  They wanted to send out a s|
|pecial, personalized letter to th|
|eir richest 1000 customers.  Unfo|
|rtunately for the programmer, he |
+----------------------------------+
```

打开后：

ex command
```
        :set linebreak
```

效果将是：

Display
```
+----------------------------------+
|letter generation program for a  |
|bank.  They wanted to send out a |
|special, personalized letter to  |
|their richest 1000 customers.    |
|Unfortunately for the programmer,|
+----------------------------------+
```

相关选项:

'breakat'指定了可以断行的字符.

'showbreak'可以指定一个字符串显示在接续显示的行的开头.

把'textwidth'设置为0可以避免自动断行.

移动可视屏幕行

"j"和"k"命令可以上下移动文本行. 这两个命令作用于长的文本行时每次移动的屏幕显示的行可能会多于1行[1].

要精确地每次只移动一个屏幕显示行，使用"gj"和"gk"命令. 对于根本无需折叠显示的行，这两个命令与"j"和"k"命令的效果一样，如果一个长行需要折叠显示，这两个命令就会只移动一个屏幕显示行.

使用下面绑定到箭头键的映射往往是更好的选择:

ex command
```
:map <Up> gk
:map <Down> gj
```

合并多行文本

如果你要把文件导入到一个如MS-Word的程序中去，那就应该每个段只占据一行. 如果你的每段文本现在都是以空行分隔的，下面的命令可以把每个段放到同一行上:

ex command
```
:g/./,/^$/join
```

看起来挺复杂. 我们一点一点来解释:

ex command
```
:g/./     一个全局命令，查找那些至少有一个字符的行.
,/^$/     指定一个范围，从当前行开始(非空行)直到一个空行.
join      ":join"命令把指定范围内的行连为一行.
```

下面一段文本在30列处断行，共8行:

[1]译注: 文本行指有一个换行符的行，屏幕显示行指在屏幕上占据一个显示行位置的行，一个太长而需要折叠到下行显示的文本行需要多个屏幕上的显示行来显示

Display
```
+----------------------------------+
|A letter generation program       |
|for a bank.  They wanted to       |
|send out a special,               |
|personalized letter.              |
|                                  |
|To their richest 1000             |
|customers.  Unfortunately for     |
|the programmer,                   |
+----------------------------------+
```

执行该命令后变为两行:

Display
```
+----------------------------------+
|A letter generation program for a |
|bank.  They wanted to send out a s|
|pecial, personalized letter.      |
|To their richest 1000 customers.  |
|Unfortunately for the programmer, |
+----------------------------------+
```

注意如果分隔段的行含有空白字符的话可不是所谓空行;上面的命令就不行了. 如果你的分隔行含有空格或制表符这些字符的话. 代之以这个命令:

ex command
```
:g/S/,/^s*$/join
```

还是需要段尾有一个特别的行来标识段的结束.

25.5 编辑表格

假设你正在编辑这面这样的一个四栏表格:

Display
```
nice table          test 1        test 2        test 3
input A             0.534
input B             0.913
```

现在需要在第3栏里输入数字. 最普通的做法是移到第二行去,使用命令"A",然后输入一些空格,对齐位置后输入你的数字.

对于这类编辑任务有一个特殊的选项:

ex command

```
set virtualedit=all
```

现在你可以把光标移到空无一物的虚位置上去了. 这叫"虚空白". 这时编辑上面这样的表格就容易多了.

通过搜索最后一栏的标题来移动光标:

normal mode command

```
/test 3
```

现在按下"j"命令光标就刚好在你要输入数字的位置上了, 假如输入了"0.693"结果将是:

Display

```
nice table          test 1      test 2       test 3
input A             0.534                    0.693
input B             0.913
```

Vim会自动填充你新输入的内容之前的空旷地带. 现在可以用"Bj"命令开始输入下一个栏位的内容了. "B"将光标移回到输入新内容之前的位置. "j"则将光标又下移一行.

> 备注: 你可以把光标移到任何位置上去, 即使是在一行行尾的后面. 但除非你真正输入了新的内容, 否则Vim不会因此就填充留出的空白.

复制一个表格的列

假设你要增加一列, 该列位于"test 1"列之前, 内容上近似于第3列. 这可以通过以下7步完成:

1. 将光标移动到该列的左上角, 比如, 用"/test 3".
2. 按下CTRL-V进入列选择模式.
3. 将光标下移两行:"2j". 现在光标已在"虚空白"上, "test 3"列在"input B"那一行的位置上.
4. 右移光标, 把整个列都选择进来, 还要加上列间距. 使用"9l".
5. 用"y"选择被选择的矩形区域.
6. 将光标移至"test 1", 新的列即将被放在这里.
7. 按下"P"命令.

结果如下:

Display
```
nice table          test 3   test 1     test 2      test 3
input A             0.693    0.534                  0.693
input B                      0.913
```

注意两个"test 1"列都被右移了，包括那些"test 3"栏位并没有内容的那些行.

下面的命令使光标移动命令的行为恢复到正常状态:

ex command
```
:set virtualedit=
```

虚替换模式

使用'virtualedit'选项的缺点是让人感觉怪怪的. 把光标移到行尾后面的时候你也不知道那里有什么，空格还是制表符? 另外还有一种替代它的办法: 虚替换模式.

假设你的表格中有下面一行，有制表符，也有其它字符. 在第一个制表符上使用"rx" 命令可能会弄乱整个内容:

Display
```
inp     0.693   0.534   0.693

       |
   rx  |
       V

inpx0.693   0.534       0.693
```

要避免这样的结果，代之以"gr"命令:

Display
```
inp     0.693   0.534   0.693

       |
  grx  |
       V

inpx    0.693   0.534   0.693
```

不同在于"gr"命令总是让被替换的文本占据它所应有的屏幕空间. 空出的间隙会以额外的空格或制表符来填充. 实际执行的就相当于把制

表符替换为字符"x"然后又增加了一些空白来让其后的内容保持在固定位置. 该例中插入了一个制表符.

使用"R"命令到replace模式下(参看|04.9|)替换多个字符时也一样, 原有的整洁布局被弄乱:

Display
```
inp     0       0.534   0.693

         |
 R0.786  |
         V

inp     0.78634 0.693
```

对应的"gR"命令使用虚替换模式. 它将保持页面的良好布局:

Display
```
inp     0       0.534   0.693

         |
gR0.786  |
         V

inp     0.786   0.534   0.693
```

下一章: |usr_26.txt| 重复重复, 再重复
版 权: 请参考|manual-copyright| vim:tw=78:ts=8:ft=help:norl:

usr_26.txt Vim 6.3版.最后修改：2002年10月29日

VIM用户手册--- 作者： Bram Moolenaar

重复重复，再重复

请问： 该标题出自星哥的哪部电影?

---周星驰专业8级试题B卷

一个编辑任务很少是杂乱无章的. 通常同一个改动需要重复多次. 本章介绍了几种重复先前的改动的方法.

下一章： |usr_27.txt| Search commands and patterns
前一章： |usr_25.txt| Editing formatted text
目　录： |usr_toc.txt|

26.1 Visual模式的重复

Visual模式非常适用于以任意顺序改变一行的内容. 你可以看到高亮的文本，可以清楚地看到是否改对了. 但是选择被操作的文本颇费时间. "gv"命令可以再次选定上次选择的Visual区域. 这使你可以对同样的文本对象施以其它的操作.

假设你想把下面的文本中所有的"2001"都改为"2002"，"2000"改为"2001"：

Display
```
The financial results for 2001 are better
than for 2000.  The income increased by 50%,
even though 2001 had more rain than 2000.
                2000            2001
income          45,403          66,234
```

先把"2001"改成"2002". 在Visual模式下选定这些行，然后用命令：

ex command
```
        :s/2001/2002/g
```

现在用"gv"命令再次选定同一文本对象. 不管光标在哪这招都管用. 然后用":s/2000/2001/g"进行第二步的修改.

显然，你可以重复重复再重复地进行修改.

26.2 加与减

需要重复地进行数字的加减时，通常加减量是一样的. 比如上例中每个年份都加1. 除了象上面一样用替换命令进行修改之外，CTRL-A命令也可以[1]

同样对上例中的内容，使用搜索命令来查找一个年份:

normal mode command

```
/19[0-9][0-9]\|20[0-9][0-9]
```

现在按CTRL-A. 当前光标下的年份会被加1:

Display

```
The financial results for 2002 are better
than for 2000.  The income increased by 50%,
even though 2001 had more rain than 2000.
                2000            2001
income          45,403          66,234
```

"n"命令查找下一个年份，"."来重复上一次的CTRL-A("."更好键入一些). 不断重复"n"和"."直到所有的年份都修改完毕.

提示: 设置'hlsearch'选项可以让你对这些要改动的地方一目了然，做起来信心也更足一些.

加减1的命令CTRL-A也可以前缀以一个数字参数. 如果你有下面的列表项:

```
1. item four
2. item five
3. item six
```

置光标于"1"上. 按下:

normal mode command

```
3 CTRL-A
```

[1]译: 替换命令还有一个唯美的方案:
:3,5s/19[0-9][0-9]\|20[0-9][0-9]/=((submatch(0)+1)/g)

"1"将变为"4". 同样你可以用"."来将这一操作施于其它数字上.

另一个例子:

Display
```
006     foo bar
007     foo bar
```

在这两个数字上使用CTRL-A的结果是:

Display
```
007     foo bar
010     foo bar
```

7加1等于10? 原因是Vim将"007"视为一个八进制的数了，因为前面有0嘛. C程序中经常会出现这种记法. 如果你不想让此类数字被看作是8进制的，可以改变下面的选项:

ex command
```
:set nrformats-=octal
```

CTRL-X以类似的方式做减法操作.

26.3 对多个文件做同样的改动

如果你想把好几个C文件中名为"c_cnt"的变量都改为"x_counter". 这就要动到多个文件.

先把所有要改的文件放到参数列表上:

ex command
```
:args *.c
```

该命令会找到所有的C文件并开始编辑第一个. 现在你可以对所有的文件都进行同一个替换操作:

ex command
```
:argdo %s/\<x_cnt\>/x_counter/ge | update
```

":argdo"命令以另一个命令为参数. 该命令将对所有待编辑的文件都执行一次.

"%s"替换操作将施于所有行上. 它通过"\<x_cnt\>"查找"x_cnt". 其中"\<"和"\>"使得只有完整的单词会被匹配，这样"px_cnt"和"x_cnt2"中的x_cnt才可以免遭毒手.

替换操作的标志"g"使得每行中的全部"x_cnt"都被替换.标志"e"则用于避免某些文件中一个"x_cnt"都找不到时的错误消息. 否则的话":argdo"命令遇到这些错误就会终止整个操作.

"|"用来分隔两个命令. 后面的"update"命令会在文件有改变时进行保存. 如果没有一个"x_cnt"被替换为"x_counter"那就不进行任何操作.

类似于":argdo", 命令":windo"会对所有窗口都执行同样的操作. ":bufdo"则是对所有的缓冲区执行操作. 这个要小心使用, 因为你可能想不到缓冲区列表中还有那么多文件.最好使用该命令之前用":buffers"命令(或":ls")看一下就有哪些缓冲区会被改动.

26.4 在一个shell脚本中使用Vim

如果你将很多文件中的"-person-"都改为"Jones"其后打印出来. 你会怎么办? 拼命地敲打键盘. 还是写一个shell脚本来让计算机自动操作?

作为一个全屏幕编辑器, Vim的Normal模式命令作出十分出色. 但对批处理任务来说. Normal模式的命令的结果是什么就不得而知了. 这里最好是用Ex模式的命令. 该模式下的命令行界面的命令很适于放入一个批处理文件中.("Ex command"只是命令行命令/冒号命令的另一说法)

要执行的Ex模式命令如下:

ex command
```
%s/-person-/Jones/g
write tempfile
quit
```

将这几行命令放入"change.vim"中. 现在以批处理模式运行运行Vim:

shell command
```
for file in *.txt; do
  vim -e -s $file < change.vim
  lpr -r tempfile
done
```

shell的控制结构for-done循环将对每个文件都施以循环体中的两行操作, 每次循环都将$file变量赋值为一个不同的文件名.

第二行的命令以Ex模式(-e参数)运行Vim来编辑文件$file, 从"change.vim"中读取要执行的命令. -s参数告诉Vim安静地运行. 也就是说, 不要再不断给出:号以及其它不必要的提示了.

"lpr -r tempfile"命令将打印"tempfile"的内容,然后删除它(-r参数的用处).

从标准输入读取

Vim可以从标准输入读取要编辑的内容。因为通常情况下它从那读取的都是命令，所以你要告诉Vim现在从标准输入读取的是编辑内容。这需要以"-"参数来代替文件名，如：

shell command
```
ls | vim -
```

该命令将"ls"命令的输出作为编辑的内容，注意此时的缓冲区没有一个对应的文件名。

如果你已经以标准输入来读取编辑内容，还可以用"-S"参数来读取脚本：

shell command
```
producer | vim -S change.vim -
```

Normal模式的脚本

如果你真的需要在脚本中执行Normal模式的命令，可以这样用：

shell command
```
vim -s script file.txt ...
```

> 备注： "-s"与"-e"连用时的意义与这里不一样。 此处它的意思是将"script"中的脚本作为Normal模式的命令执行。与"-e"连用时它意思是silent(安静)，并不会把下一个参数作为要执行的脚本文件。

"scrip"中的命令将象你手工键入一样执行。 别忘了按下回车键时换行符会被Vim接管下来解释。 在Normal模式它的作用是下移一行。

创建这样的脚本你当然可以一个字符一个字符键入。 但是可以想象这有多麻烦。另一个办法是在执行这些命令的同时把它们记录下来。 象这样：

shell command
```
    vim -w script file.txt ...
```

所有的按键都会被记录到"script"中。 如果编辑过程中出了点岔子你也可以等下手工编辑脚本文件。

"-w"参数将把新键入的命令追加到脚本文件的最后。 你想一点一点累积这些编辑记录的话这倒是不错。什么时候想全部重新开始时就用"-W"参数，它会覆盖该文件的内容。

下一章：	|usr_27.txt| 搜索命令和模式语言
版　权：	请参考|manual-copyright| vim:tw=78:ts=8:ft=help:norl:

usr_27.txt Vim 6.3版.最后修改: 2003年10月28日

VIM用户手册--- 作者: Bram Moolenaar

搜索命令和模式语言[1]

在第3章介绍了一些简单的搜索模式. Vim可以执行远远比这复杂得多的搜索. 本章讲解这些最常用的搜索. 关于模式搜索更详细的话题可以参考|pattern|.

下一章: |usr_28.txt| 折行
前一章: |usr_26.txt| 重复执行
目 录: |usr_toc.txt|

27.1 忽略大小写

默认情况下Vim的搜索是大小写敏感的. 这样"include", "INCLUDE"和"Include"就是三个不同的word, 一次Vim搜索只匹配它们中的一个.

现在打开'ignorecase'选项:

ex command
```
:set ignorecase
```

再搜索一下"include"看, 现在它也可以匹配到"Include", "INCLUDE"和"InClUDe"了. (建设设置'hlsearch'选项以快速浏览就有哪些地方符合匹配).

下面的命令又可以关闭这一选项:

[1]译者: 正则表达式是vim中的重头戏, 本手册中的regular expresion, pattern一般情况下可以互用, 都是指这种格式紧凑面目可憎的神秘符号. 用于以抽象的方式指定一类要搜索/替换的对象

ex command
```
:set noignorecase
```

但这里我们还是暂且设置该选项，查找"INCLUDE". 这样它就同样可以匹配到"include".现在打开'smartcase'选项:

ex command
```
:set ignorecase smartcase
```

如果你要搜索的内容中至少包括一个大写字母，整个搜索就会是大小写敏感的. 这样设计你就不必总是输入大写字符了，你想要进行大小写敏感的搜索时准确键入就行了. 这看起来智能多了.

设了上面这两个选项，下面的所有word都可以搜索得到:

List

pattern	matches
word	word, Word, WORD, WoRd, etc.
Word	Word
WORD	WORD
WoRd	WoRd

正则表达式内部的大小写

如果你只想对搜索模式的一部分应用大小写不敏感的策略，可以在它前面加上一个"\c". 使用"\C"会使大小写敏感. 而且这两个前缀的优先级高于'ignorecase'和'smartcase'选项的设定，使用"\c"或"\C"时Vim不会考虑这两个选项的值是什么.

List

模式	能匹配什么
\Cword	word
\CWord	Word
\cword	word, Word, WORD, WoRd, etc.
\cWord	word, Word, WORD, WoRd, etc.

使用"\c"和"\C"还另有一大优点，它们与要搜索的模式字符串写在一起，这样你从一个搜索命令历史列表中调出它们时，还可以精确地使用大小写敏感或不敏感[1]，不管'ignorecase'的'smartcase'的设置是否已经被改变，搜索的结果都会与上次使用它们时保持一致.

> 备注: 在搜索模式中对"\"的解释因'magic'选项的设置而异. 本章中我们都假定'magic' 选项是打开的，因为这是标准的情况，也是我们推荐的设置. 如果你改变了'magic'，可能很多本身好好的搜索模式都会不对劲了.

[1]译注: 或是一部分敏感另一部分不敏感

> 备注: 如果搜索的执行花了太长时间还没有返回,你可以中途打断它. 在Unix上用CTRL-C,在MS-DOS和MS-Windows上用CTRL-Break.

27.2 绕回文件头尾

默认情况下,一个向前的搜索会从当前光标开始,直到找到目标或达到文件尾. 如果到了文件尾还没有找到匹配的字符串,它就会从文件开头继续查找.

记住每次你用"n"命令来重复进行搜索时,你有可能实际上是回到了第一个匹配处.如果你从来都没有注意过的话,也可以注意一下Vim给你的提示:

Display
```
search hit BOTTOM, continuing at TOP
```

如果你用"?"命令,从反方向开始搜索,得到的提示是:

Display
```
search hit TOP, continuing at BOTTOM
```

如果你还是不知道什么时候你又回到了第一个匹配. 还可以试试这样办法,打开'ruler' 选项:

ex command
```
:set ruler
```

Vim会在窗口的右下角处显示当前的光标位置(如果有状态行的话它会出现在状态行上). 看起来象这样:

Display
```
101,29        84%
```

第一个数字是当前所在的行号. 记住你从哪一行开始第一次搜索的,这样你就可以对比检查什么时候越过了这个位置.

不循环搜索

要关闭越过文件头尾的循环搜索,可以使用下面的命令:

ex command
```
:set nowrapscan
```

现在如果搜索已经达到了文件尾就会显示如下的错误信息:

Display
```
E385: search hit BOTTOM without match for: forever
```

这样你可以以下面的方法找到所有的匹配: 用"gg"命令回到文件的开头，继续搜索直到再次看到这样的错误信息.

如果你是在用"?"做反方向搜索，你看到的将是会:

Display
```
E384: search hit TOP without match for: forever
```

27.3 偏移

默认情况下，搜索到一个目标后光标会停留在目标字符串的第一个字符上. 你也可以告诉Vim此时要如何放置光标. 对于向前搜索命令"/"来说，指定一个偏移可以通过在模式最后再加上一个/符号，紧接着指定你要的偏移.

normal mode command
```
/default/2
```

这个命令会搜索模式"default"，找到后将光标停留在目标行向下的第2行. 对上一段执行这个命令的话，Vim将会在第1行发现"default". 然后光标置于其后的第2行即"an offset"上[1]..

如果指定的偏移是一个简单的数字，光标就会被简单地置于目标行其下第N行的开头. 偏移可以是正的也可以是负的. 如果是正的的话，光标会在找到目标行后向前移动，反之向后移.

微调查找到结果时的光标位置

"e"指示光标在找到目标串之后以它的结尾作为移动的起始处. 它会把光标置于目标字符串的最后一个字符，命令:

normal mode command
```
/const/e
```

会在找到"const"后把光标置于"t"上.

从光标的默认位置上附加一个数字将引起光标向前移动指定的字符数，下面的命令将光标置于目标字符串的结尾处的下一个字符:

[1]译注: 这段话只对英文有效，对于中文你可以搜索"默认"，同样的命令会让你停留在开头是"模式" 所在的那一行

ex command
```
/const/e+1
```

正数使光标向右移动，负向左移． 如：

ex command
```
/const/e-1
```

会将光标移到"const"的"s"上．

如果偏移指示符是以"b"开始的，则光标移动到目标字符串的开头．这个用处不大，因为默认就是如此． 只有在加减一个数时它才显得有用．此时光标从目标字符串的开头左右偏移． 如：

ex command
```
/const/b+2
```

命令会以目标字符串的开头为原点，向右移动两个字符，最终停留在"n"上．

重复

要重复前一个搜索但应用不同的偏移，可以空出模式部分：

ex command
```
/that
//e
```

它与：

ex command
```
/that/e
```

完全一样．

如果要用的偏移也是一样，用

ex command
```
/
```

就行．

"n"命令与此相同．要继续查找但移除前面使用的偏移效果，用：

ex command
```
//
```

反向搜索

对"?"命令而言使用偏移也毫无二致. 但是你必需以"?"来分隔命令的不同部分:

ex command
```
?const?e-2
```

"b"和"e"还保持原来的意思, 它们不因"?"命令的方向而改变.

开始位置

搜索命令通常始于当前位置. 如果你同时指定了一个行偏移, 这就可以引起问题. 比如:

ex command
```
/const/-2
```

这个命令查找下一个"const"然后定位于其上的第2行. 如果你用"n"命令继续搜索, Vim就会从现在的位置搜索下一个"const". 应用同样的偏移, 你又回到了刚才的那个地方. 你会一直被耗在同一个位置!

这可能会引起错误: 假设下面还有一个"const"符合匹配. 继续向前查找就会找到这个"const"然后又向上移2行. 这样实际上你的光标是往后移了!

当你指定的是以字符为单位的偏移时, Vim会为此作出补偿. 这样同样的目标字符串就不会被再次匹配到.

27.4 多次匹配

"*"在一个模式中表明它前面的项可以匹配任意次数. 这样:

ex command
```
/a*
```

就可以匹配到"a", "aa", "aaa", 等等. 但是它还能匹配""(空字符串), 因为0次匹配也包括在内.

不过"*"只对紧靠在它前面的项起作用. 所以"ab*"匹配的是"a", "ab", "abb", "abbb", 等等这样的东西. 要让整个字符串重复多次, 必需让它们成组作为一个项. 用"\(" 和"\)"把它们前后包围起来即可. 下面的命令:

ex command
```
/\(ab\)*
```

就可以匹配到: "ab", "abab", "ababab", 等等. 还有"".

要避免匹配到一个空的字符串, 使用"\+". 它让前面的项重复1次或多次.

normal mode command
```
/ab\+
```

会匹配到"ab", "abb", "abbb", 等等, 但不会匹配后面没有一个"b"的"a".

要匹配一个可有可无的项, 使用"\=". 例如:

normal mode command
```
/folders\=
```

匹配"folder"和"folder".

指定重复次数

要指定重复的次数, 可以使用"\{n,n}"这样的形式. "n" 和"m"代表数字. 其前的项会被匹配"n"次到"m"次. 参考|inclusive|. 例:

normal mode command
```
/ab\{3,5}
```

匹配"abbb", "abbbb" 和"abbbbb".如果"n"被忽略了就默认它为0, 如果"m"被忽略了就默认它为无穷大.如果",m"被忽略了那就会精确地匹配"n"次重复.

List

模式	匹配到的次数
\{,4}	0, 1, 2, 3 或 4次
\{3,}	3, 4, 5次, 等等.
\{0,1}	0 或 1, 跟 \= 一样
\{0,}	0 次或多次, 跟 * 一样
\{1,}	1 次或多次, 跟 \+ 一样
\{3}	3 次

正则表达式: 懒惰模式[1]

[1]译: 在Jeffrey E.F. Friedl经典的<<Mastering Regular Expressions>>一书中, 将正则表达式默认的工作方式---匹配尽可能多的内容称为greedy(贪婪)模式, 而将匹配尽可能少的内容这种工作方式称为lazy(懒惰模式), 这也是perl中对这两种模式的标准术语

目前为止，所有的重复项都是"贪婪"地匹配所能找到的字符．要尽可能少次数地重复一个项，使用"\{-n,m}"． 它跟"\{n,m}"一样，只是在匹配时尽可能少次数地重复．

例如，用命令：

normal mode command
```
/ab\{-1,3}
```

将会匹配到"abbb"中的"ab"． 实际上，它永远都不会匹配多于一个的b，因为没理由做这样的匹配． 要让它超出最低限定次数地重复必需要有其它的强制因素．

对"n"或"m"一方缺角的情况也一样． 甚至两个上下限都没有指定时也一样，如"\{-}".它匹配它前面的项一次或0次，尽可能地少． 这个模式本身只可能匹配到0次． 跟其它东西联合使用时这一功能十分有用．如：

normal mode command
```
/a.\{-}b
```

它会匹配到"axbxb"中的"axb"． 如果模式是：

normal mode command
```
/a.*b
```

它就会尽可能多地匹配了． 所以匹配到的是整个"axbxb"．

27.5 多选一

在一个模式中的"或"操作符是"\|"． 如：

normal mode command
```
/foo\|bar
```

它匹配到"foo"或者"bar"． 更多的并列项可以继续串联在一块：

normal mode command
```
/one\|two\|three
```

匹配到"one"，"two" 和"three"．

要匹配多次，必需把整个字符串用"\("和"\)" 前后括起来：

normal mode command
```
/\(foo\|bar\)\+
```

这可以匹配到"foo"，"foobar"，"foofoo"，"barfoobar"，等等.

看另一个例子：

normal mode command
```
/end\(if\|while\|for\)
```

匹配的是"endif"，"endwhile" 和"endfor".

另一个与此相关的项是"\&". 它要求两个并列的选项同时被被匹配到. 最终的匹配结果将是最后一个并列项. 如：

normal mode command
```
/forever\&...
```

将只会匹配"forever"中的"for". 但不会匹配到"fortuin"中的"for"[1].

27.6 字符范围

要匹配"a"或"b"或"c"你可以用"/a\|b\|c". 如果你要匹配的是从"a"到"z"的所有26个字母这个模式就会变得很长很长... 下面是另一种更为简短的表示法：

normal mode command
```
/[a-z]
```

[]这种结构只匹配到一个单个的字符. 在括号中你可以指定哪些字符可以被匹配到. 你可以指定一个字符列表，象这样：

normal mode command
```
/[0123456789abcdef]
```

这将会匹配到所有包括在内的单个字符. 对于ASCII以1递增的连续字符你可以指定一个范围. "0-3"代表"0123". "w-z"代表"wxyz". 所以上面的整个命令可以写为：

normal mode command
```
/[0-9a-f]
```

要匹配一个"-"本身只需把它放在整个字符集的开头或结尾. 下面这些特殊字符可以在[]中出现(实际上它们可以出现在一个搜索模式的任何地方)：

[1]译注： 因为"fortuin"不符合并列项中的forever，在forever&...中，首先forever这个被匹配到，然后3个任意字符一定会被匹配到，实际上就是for，最终被匹配的是for，所以如果搜索命令是/forever&.../e则光标将停留在字母for中的r上. 关于\&，请参考|/\&|

List

```
e        <Esc>
\t       <Tab>
\r       <CR>
\b       <BS>
```

关于[]的范围还有一些特别的情况，参考|/[]|可以获知该主题的完整内容.

补集

要避免匹配到某个特殊的字符，在[]字符集的开头用"^"可以指定除[]中指定的所有字符之外的字符. 如下:

List

```
/"[^"]*"
 "          一个双引号
  [^"]      除双引号外的任何字符
      *     尽可能多地匹配
       "    又一个双引号
```

这将会匹配到"foo"和"3!x"，包括双引号本身.

预定义字符集

由于字符集合在Vim中被广泛使用. 所以Vim提供了另一种快捷的表示:

normal mode command

```
/\a
```

查找所有的字母字符. 等同于"/[a-aA-Z]". 下面是其它一些类似的类集表示:

List

特殊项	匹配什么	等价的正则表达式	
\d	数字	[0-9]	
\D	非数字	[^0-9]	
\x	十六进制数	[0-9a-fA-F]	
\X	非十六进制数	[^0-9a-fA-F]	
\s	空白字符	[ ]	(<Tab> 和 <Space>)
\S	非空白字符	[^]	(除 <Tab> 和 <Space>之外)
\l	小写字母	[a-z]	
\L	非小写字母	[^a-z]	
\u	大写字母	[A-Z]	
\U	非大写字母	[^A-Z]	

备注: 使用这些预定义的类集会比手工在[]中指定一个等价的类集要快得多[a]. 这些项不能用于[]内部. 所以"[\d\l]"并不会象你想象的那样去匹配数字或小写字母. 用"\(\d\|\l\)"就OK了.

[a]译注: Vim在内部已编译过这些类集所对应的内部形式, 而临时指定的类集需要即时编译

参看|/\s|以了解特殊类集的完整列表.

27.7 字符分类

字符类集可以匹配一个数目固定的字符集合. 字符类与此相似, 但是有一点本质不同: 字符类中的元素可以在不改变搜索模式的情况下重新定义.

比如查找下面的模式:

normal mode command

```
/\f\+
```

"\f"项代表组成文件名的字符. 这个模式可以匹配到一个看似文件名的字符串.

一个合法的文件名到底可以含有哪些字符依具体的操作系统而定. 在MS-Windows上, 可以包含一个反斜杠, 在Unix上就不能. 这可由'isfname'选项指定. 该选项在Unix上的默认值是:

ex command

```
:set isfname
isfname=@,48-57,/,.,-,_,+,,,#,$,%,~,=
```

对其它系统来说这个默认值就不同了. 所以你可以用"\f"来搜索一个文件名, 不过它应该被预先调整得适宜于你所工作的系统.

备注: 实际上, Unix可以允许包括空格在内的任何字符. 在'isfname'选项里包含这些字符在理论上也是正确的[a]. 但是这样的话Vim就没办法判断一个文件名到底在哪里结束了. 所以默认的'isfname'选项的采取了实用的热衷主义[b].

[a]译注: 实际中也是正确的, 但是会造成诸多不便

[b]译注: 实际上Unix系统的文件名可以是任何除'0'和'/'之外的字符, 前者用于标记一个字符串的结束, 后者用来分隔一个路径中不同的部分

这些字符类还包括[1]:

List

项	匹配什么	对应的选项
\i	标识符字符	'isident'
\I	同于\i，但排除了数字	
\k	关键字字符	'iskeyword'
\K	同\k，但排除数字	
\p	可打印字符	'isprint'
\P	同\p，但排除数字	
\f	文件名字符	'isfname'
\F	同\f，但排除数字	

27.8 匹配一个断行

Vim可以查找下一个包括断行符号的字符串. 你需要告诉它在哪里断行，因为目前为止所有的项都不包含断行符[2].

要表明某处发生断行，使用"\n":

normal mode command
```
/the\nword
```

这将会匹配到以"the"结束而且下一行以"word"开始的行. 要同时匹配"the word"，你需要匹配空格或断行. "_s"项正是这个的意思:

normal mode command
```
/the\_sword
```

[3]下面的例子允许多个的空白字符:

normal mode command
```
/the\_s\+word
```

这会同时匹配到行尾是"the "下一行开头是" word"的情况.

"\s"匹配空白，"_s"匹配空白或断行. 同样，"\a"匹配一个字母字符，"_a"匹配一个字母字符或一个断行. 其它的字符类也一样.

还有很多其它的项都可以通过前缀以"_"来同时包括断行. 比如"_."可以匹配包括断行在内的任何字符.

[1]译注: 规律: 命令的大写形式是对小写形式的某种修饰，此例中减去一个集合

[2]译注: 这里用断行符是因为在不同的系统上换行的标识不一致，MS-DOS和MS-Windows是用"回车"(ASCII为13)和" 换行"(ASCII为10)两个连续的字符来表示，Unix系统用一个"换行"(ASCII为10)来表示，Mactonish则用"换行" (ASCII为10)和"回车"(ASCII为13)两个连续的字符来表示

[3]译注: 在一个表示字符类的项中\后面附加一个_表示为这个字符类再附加一个元素:断行

备注: "_.*"匹配到行尾的所有东西. 用这样的命令要谨慎一点, 它可能会很慢.

另一个例子是"_[]", 它同样可以让一个字符类集额外地包含一个断行[1]:

normal mode command
```
/"\_[^"]*"
```

这个命令可以查找双引号引起来的字符串, 即使它们跨过了行边界.

27.9 例子

现在是一些搜索模式的例子. 它们展示了如何组合使用上面的这些技巧.

查找一个加州的汽车牌照

一个汽车牌照的样例如"1MGU103". 它包含一个数字, 3个大写字母和接下来的3个数字. 可以把它们直接反映在一个模式中:

normal mode command
```
/\d\u\u\u\d\d\d
```

另一个办法是指定字母或数字的个数:

normal mode command
```
/\d\u\{3}\d\{3}
```

用[]字符类集的话是:

normal mode command
```
/[0-9][A-Z]\{3}[0-9]\{3}
```

你喜欢哪一种方法? 不管你首先想起哪一个, 你最先想起来的就是最简单的. 如果你同时知道上面的所有写法, 那么最好不要用最后一种, 因为它要键入更多的字符而且执行起来也很慢[2]

查找标志符

在C程序中(对很多其它的计算机语言也是如此)一个标识符总是以一个字母开头, 后面是字母或数字. 下划线也可用于标识符的开头. 这可以用下面的模式来识别:

[1]译注: 也可以用[...\n]

[2]译注: 如果你能记得所有这些写法, 你真是一个正则表达式专家, 那么你也就不需要这样的忠告, 因为作为专家你也一定知道如何鉴定正则表达式的优劣;-).

normal mode command

```
/\<\h\w*\>
```

"\<"和"\>"或用于识别整个的单词. "\h"代表"[A-Za-z_]", "\w"代表".[0-9A-Za-z]" [1]

备注: "\<"和"\>"的工作视'iskeyword'选项的值而定. 如果它包含了"-", 比如"ident-"这样的情况就不会被匹配[a]. 这时可以用: \w\@<!\h\w*\w\@!这个模式检查"\w"即不出现在一个标识符的前面, 也不出现在其后面.参考/\@<!和/\@!.

[a]译注: 这里的意思比较令人费解, 为什么'iskeyword'多了一个元素反而匹配不到了呢? 如"ident-a"这样的一个字串, "\<\h\w*"可以匹配到"ident", 这里\w不包括"-"所以不能匹配到"ident-", 但恰恰此时"-"是一个所谓keyword, 这样后面的"\>"就满足不了了, 所以整个模式匹配失败

下一章: |usr_28.txt| 折行

[1]译注: 不幸ASCII表的排列中9与A, Z与a都是不连续的, 不然你可以写"[0-z]"

usr_28.txt Vim 6.3版.最后修改: 2003年12月21日

VIM用户手册--- 作者: Bram Moolenaar

折行

结构化的内容可以划分为节，每节又可以划分小节. 折行功能可以将一节浓缩为一行，只显示其大概. 本章讲解折行的用法.

下一章: |usr_29.txt| 之于程序
前一章: |usr_27.txt| 搜索命令和模式语言
目　录: |usr_toc.txt|

28.1 使用何种折行方法?

折行用于把缓冲区中的多行文本仅显示为一行. 就象把一张纸折叠起来一样:

Display

```
+------------------------+
| line 1                 |
| line 2                 |
| line 3                 |
|_______________________ |
\                        \
 \________________________\
 / folded lines           /
/________________________/
| line 12                |
| line 13                |
| line 14                |
+------------------------+
```

被折起的内容还在缓冲区中，丝毫未变．折行只影响文本在屏幕上的显示．

使用折行你可以得到文件的一个大纲． 把一节的内容折起显示为一行，该行会标识出这里是一个折行．

28.2 手工折行

试一下把光标置于一段中使用下面的命令：

normal mode command
```
zfap
```

你将看到整个一段的内容被一个高亮的行所取代．你已经创建了一个折行． |zf|是一个操作符命令，|ap|是一个文本对象．你可以用|zf|来搭配任何的位移命令创建折行． |zf| 也可以在Visual模式使用[1]．

要再次查看折叠起来的文本，可以使用命令：

normal mode command
```
zo
```

还可以用下面的命令重新折起：

normal mode command
```
zc
```

所有与折行相关的命令都以字符"z"打头．因为它看起来正象是把一张纸折叠起来的样子． "z"之后的字符是一个易于记忆的命令：

normal mode command
```
zf        创建折行
zo        打开折行
zc        关闭折叠
```

折行可以是嵌套的： 一段包含了折行的文本还可以再折叠起来． 比如你可以把本节中的每一段都折叠起来，然后把本章中的所有节再折叠起来． 试一试．你会发现打开本章的折叠的同时恢复了被嵌套在内的那些折行，这些嵌套的折行将保持它们被更大的折行折叠起来之前的状态，或开或闭．

假设你创建了几层深的折行，现在想查看所有的文本，你可以逐个用命令"zo"打开它． 要更快的办法，试一下命令：

[1]译者： zf命令搭配的位移命令移动不出当前行时当然没有效果

normal mode command
```
zr
```

该命令会R-educe[1]折叠的层次. 相反的命令是:

normal mode command
```
zm
```

该命令是折叠得更多. 你可以重复使用"zr"和"zm"来打开或关闭多层嵌套的折行.

如果你嵌套了多层的折行,也可以用这个命令一次打开所有折行:

normal mode command
```
zR
```

这个命令减少折行的嵌套深度直到穷尽所有的折行. 下面的命令则可以关闭所有的嵌套折行:

normal mode command
```
zM
```

你可以用|zn|命令来禁用一个折行. 然后用|zN|还可以恢复它. |zi|命令则可以在两者之间切换. 这是一种有效的工作方式:

- 创建折行进行大纲预览
- 移动到某处进行编辑
- 使用|zi|打开文本进行编辑
- 编辑完毕后再用|zi|打开折行进行移动

关于手工折行的详细内容请参考|fold-manual|.

28.3 使用折行

折行折叠起来时,象"j"和"k"这样的移动命令就会视之为单行一跃而过. 这可以在折起的文本间快速移动.

你可以象对待单行文本一样对它进行复制,删除和粘贴. 这在你希望重新安排程序中的函数顺序时十分有用. 首先选择正确的方法进行折叠: 'foldmethod',把每个函数定义为一个折行折叠起来. 然后用"dd"删除函数,移动光标至某处然后用"p"命令粘贴. 如果该函数还有一些行没有被折行包括进来,你可以用Visual 模式进行选择:

[1]译者: 减少

- 将光标置于要移动文本的第一行上
- 按"V"进入Visual模式
- 将光标移动到要移动文本的最后一行
- 用"d"命令删除选择的行
- 移动光标到新的位置然后用"p"命令粘贴文本

有时很难记住折行的位置，这样你就不知道光标在哪里|zo|才能打开一个折行． 下面的选项设置可以方便你查看折行：

ex command
```
:set foldcolumn=4
```

这会在窗口左边另辟出一小列空间来标识折行． "+"标识一个折叠起来的折行． "-"标识打开的折行，折叠区中的每行前面以"|"标识．

你可以通过鼠标单"+"来打开一个折行，单击"-"或"|"打开折叠．

打开所有折叠请参考|zO|．

关闭所有折叠请参考|zC|．

删除当前行的折叠请参考|zd|．

删除当前行的所有折叠请参考|zD|．

在Insert模式下，当前行的折行总是打开的，这样你可以看清输入的内容！

在折行上左右移动光标时折行会自动打开． 例如"0"命令会打开当前行的折行（如果'foldopen'选项包含"hor"的话，这也是默认值）．'foldopen'选项可以定义什么命令会打开折行． 如果你想让当前行下的折叠总是打开的，请这样设置：

ex command
```
:set foldopen=all
```

警告： 这样你将无法将光标定位到一个关闭的折行上去． 一般来说你只会临时这样做，回到默认的设置可以用命令：

ex command
```
:set foldopen&
```

你也可以这样设置来让光标离开时折行就会自动关闭：

ex command
```
:set foldclose=all
```

这会使当前行之外的所有折行都重设'foldlevel'.最好使用|zm|和|zr|来增减折行.

折行是局部于窗口的. 这样就可以对同一个缓冲区打开两个窗口,一个用折行一个不用折行. 或者一个关闭所有折行,一个打开所有折行.

28.4 保存和恢复折行

你放弃一个文件时(比如转而去编辑另一个文件),折行的状态会丢失. 下面的命令可以保存折行的定义:

ex command
```
:mkview
```

这将会保存所有折行的定义以及其它一些影响该文件外观显示的选项. 你可以通过设置'viewoptions'来控制将哪些信息保存在视图文件中. 稍后回到该文件时,你可以这样找回刚才的感觉:

ex command
```
:loadview
```

你最多可以为一个文件保存10个视图. 比如把当前的设置存为第3个视图,然后载入第2个视图:

ex command
```
:mkview 3
:loadview 2
```

注意你增删一些文字时视图可能会被破坏掉. 同时请参考'viewdir'的设置,它控制视图文件存储的路径. 你可能需要不时删除一些老旧的视图.

28.5 根据缩进的折行

通过|zf|手工定义折行太费事了. 如果你的文件结构使更细节化的内容总是有更多的缩进,你就可以让Vim根据缩进量来自动决定折行. 它会把具有相同缩进量的行作为一个折行. 缩进量更大的行会被作为嵌套的折行. 这对很多编程语言都适用.

试一下这样设置'foldmethod'的效果:

ex command
```
:set foldmethod=indent
```

然后你可以用|zm|和|zr|命令来增减折行的层次. 从下面的样例文件中就很容易看出效果:

Display
```
This line is not indented
        This line is indented once
                This line is indented twice
                This line is indented twice
        This line is indented once
This line is not indented
        This line is indented once
        This line is indented once
```

注意缩进量与折行嵌套深度的关系由'shiftwidth'来控制. 每一个'shiftwidth'的缩进量都会增加一级折行深度. 这叫折行的层级.

使用|zr|和|zm|命令实质上只是增减了'foldlevel'选项的值. 该选项也可以直接设置:

ex command
```
        :set foldlevel=3
```

这样所有3次及3次以上'shiftwidth'缩进量的折行都会被关闭. 折叠的层级设得越低. 就有越多的折行被关闭. 'foldlevel'为0时, 所有的折行都会被关闭. |zM|命令将'foldlevel'选项设为0. 相反的|zR|命令则把'foldlevel'选项设为当前文件中的最大有效值.

这样就有两种方法可以打开和关闭折行:

(A) 通过设置折叠层级.
  这使你可以快速的抽取文件的骨架以了解其结构, 或者迅速移动光标, 又可以方便地打开文本.

(B) 通过使用|zo|和|zc|命令来关闭某个折行.
  这两个命令将只针对当前的折行, 对其它的折行没有影响.

两者也可以结合使用: 首先用|zm|命令来关闭所有折行, 然后用|zo|打开个别的折行.或者反过来, 以|zR|来打开所有折行. 再以|zm|命令叠起某个特定的折行.

但是'foldmethod'被设为"indent"时手工的折叠操作将被禁用, 因为这样的话会弄乱缩进量与折叠层级的关系.

关于根据缩进量的折叠更多的内容请参考|fold-indent|.

28.6 根据标记的折行

在文本中使用特定记号可以精确标识折行的起止范围．不过缺点是这会影响文本本身的内容．

试一下命令：

ex command
```
:set foldmethod=marker
```

示例文本如下，这是一段C程序：

code
```
/* foobar () {{{ */
int foobar()
{
        /* return a value {{{ */
        return 42;
        /* }}} */
}
/* }}} */
```

注意折起后的行会显示标记之前的文本． 这样可以清楚地显示当前的折行包含了什么内容．

如果在编辑过程中因为删除了折行的结束标记可就不妙了． 好在这个问题可以通过加了标号的标记来解决，如：

code
```
/* global variables {{{1 */
int varA, varB;

/* functions {{{1 */
/* funcA() {{{2 */
void funcA() {}

/* funcB() {{{2 */
void funcB() {}
/* }}}1 */
```

每一个加了标号的标记都表明这是指定深度的折行的起始处． 这会自动结束前面一个更高层级的折行． 通过这种办法只用起始标记就可以定义所有的折行． 只有你想明确定义一个折行在何处结束时才去加上一个结束标记．

关于使用标记进行折行的更多内容请参考： |fold-marker|．

28.7 根据语法的折行

对每种不同的计算机语言Vim都分别有一个语法文件与之对应. 该文件定义了文件中不同语法项的颜色. 如果你是在Vim中阅读本文. 所用的终端支持彩色显示的话, 你现在看到的颜色正是由"help"这个语法文件定义的.

在语法文件中也可以为语法项的定义加入"fold"参数. 这样可以定义一个折行的区域. 这需要写一个语法文件, 把这些相关的定义加进去. 做起来当然有一点难度, 不过一旦搞定, 所有的折行就可以由Vim自行决定了!

本文中我们假设你正在使用一个现成的语法文件. 这样就不用过多解释了. 你可以直接用上面介绍的命令来打开或关闭折行. 编辑文件的过程中折行也会随着内容的变化自生自灭.

关于根据语法的折行更多内容请参考: |fold-syntax|.

28.8 根据表达式折行

这个类似于根据缩进量的折行, 但是它决定折行的依据不是缩进量, 而是用一个函数来计算某行的折行层深. 你可以用这种办法来决定文本中哪些行属于同一类内容. 这里的例子来自一段e-mail的下文, 被引用的文本以行首的">"来标识. 下面的命令根据引用的层次来进行缩进:

ex command
```
:set foldmethod=expr
:set foldexpr=strlen(substitute(substitute(getline(v:lnum),'\\s','',\"g\"),'[^>
```

可以在下面的文本上试一试:

Display
```
> quoted text he wrote
> quoted text he wrote
> > double quoted text I wrote
> > double quoted text I wrote
```

下面对上例中'foldexpr'的用法逐一解释:

List
```
getline(v:lnum)                    得到当前行
substitute(...,'s','','g')         删除所有的空白字符
substitute(...,'[^>].*','',''))    删除打头的'>'后面的所有东西
strlen(...)                        计算字符串的长度, 即'>'出现的次数
```

注意在":set"命令中, 每个空格. 双引号或反斜杠的前面都必需有一个反斜杠. 如果你迷惑不解的话, 用命令

```
ex command
:set foldexpr
```

检查一下这样设置的结果是什么. 要修改一个复杂的表达式，最好用下面的命令行补齐功能:

```
ex command
:set foldexpr=<Tab>
```

其中的<Tab>表示此处要你真正输出一个制表符. Vim会填补foldexpr的当前值，然后你可以在此基础上进行修改.

表达式太过复杂时你应该用一个函数把它包裹起来，然后设置'foldexpr'来调用这个函数.

关于根据表达式的折行更多的内容可以参考: |fold-expr|.

28.9 折叠没有修改的行

这对在同一窗口中设置了'diff'选项时非常有用. |vimdiff|命令会把一切都准备就绪，如下:

```
ex command
setlocal diff foldmethod=diff scrollbind nowrap foldlevel=1
```

对显示了不同版本的各窗口都以此命令进行设置，你会很清楚地看出不同版本的差异，相同的部分都会被折起.

更多细节请参考|fold-diff|.

28.10 使用何种折行方法?

太多的选择反而让人无所适从. 世上也没有绝对的真理. 这里仅提供一些提示.

如果你正在编辑的源文件有一个语法文件，那往往就是最佳选择. 如果没有，你也可以自己写一个. 这需要你熟练掌握正则表达式. 不容易! 但是想想它的好处，一旦弄好了你就再也不需要手工去定义折行了.

对于非结构化的内容可以手工定义折叠区域. 然后用|:mkview|命令来保存和恢复定义的折行.

使用标记来决定折行需要你改变文本的内容. 如果你要跟别人共享文件或者要遵循公司的文档规范，可能你就不能那样做.

标记折行法的最大好处就是你可以精确地定义你要的折行区域. 即使你增删了折叠区域中的某些行. 另外你还可以添加一段注释来说明当前折行中的内容是什么.

缩进折行法适用于很多文件, 但是效果并不总是最好的. 没有更好的办法时也只能退而求其次了.而且, 它对于归纳文件的大纲很有用. 你可以对每一个'shiftwidth' 的缩进设置一级折叠. 表达式折行法几乎适用于所有结构化的文件. 使用起来也很简单, 尤其是折行的起止行很容易识别时.

如果你用"expr"来定义折行规则, 但是效果并不如你所愿[1], 你可以切换到手工折行来. 原来的折行定义并不会因此消失. 这样你可以手工调整折行.

下一章: |usr_29.txt| 之于程序
版 权: 请参考|manual-copyright| vim:tw=78:ts=8:ft=help:norl:

[1]译者: 很可能是因为你的正则表达式不过关

usr_29.txt Vim 6.3版.最后修改: 2004年02月12日

VIM用户手册--- 作者: Bram Moolenaar

之于程序

Vim的作者就是一个程序员[1].所以不奇怪Vim拥有大量辅助写程序的功能:辗转于标识符的定义语句与引用语句. 在另一个窗口中预览程序中相应的声明部分,等等. 下一章还会介绍更多这方面的东西.

下一章: |usr_30.txt| 编辑程序
前一章: |usr_28.txt| 折行
目 录: |usr_toc.txt|

29.1 使用tags

tag是什么? 一个位置. 记录了关于一个标识符在哪里被定义的信息. 比如C或C++程序中的一个函数定义. 这种tag聚集在一起被放入一个tags文件. 这个文件可以让Vim能够从任何位置起跳达到tag所指示的位置--标识符被定义的位置.

下面的命令可以为当前目录下的所有C程序文件生成对应的tags文件:

shell command
```
ctags *.c
```

"ctags"是一个独立的程序. 绝大多数Unix系统上都会预装这个程序,如果你的系统上还没有,你可以在下面找到一个功能丰富的ctags:

URL
```
http://ctags.sf.net
```

[1]译者: 这句大白话翻译得有点...让我想起里根的一句名言: 美国是一个大国,人民主要是美国人

现在你在Vim中要跳到一个函数的定义就可以用下面的命令：

ex command
```
:tag startlist
```

这个命令会带你到函数"startlist"的定义处，哪怕它是在另一个文件中.

CTRL-]命令会取当前光标下的word作为tag的名字并直接跳转. 这使得在大量C程序中进行探索更容易一些. 假设你正看函数"write_block"，发现它调用了一个叫"write_line"的函数，这个函数是干什么的呢？你可以把光标置于"write_line"上，按下CTRL-]即可.

如果"write_line"函数又调用了"write_char".你当然又要知道这个函数又是什么功能. 同时，置当标于"write_char"上按下CTRL-]. 现在你位于函数"write_char"的定义处.

code
```
+-------------------------------------+
|void write_block(char **s; int cnt)  |
|{                                    |
|   int i;                            |
|   for (i = 0; i < cnt; ++i)         |
|      write_line(s[i]);               |
|}      |                             |
+-------|-----------------------------+
        |
 CTRL-] |
        |  +----------------------------+
        +->|void write_line(char *s)    |
           |{                           |
           |   while (*s != 0)          |
           |      write_char(*s++);     |
           |}      |                    |
           +-------|--------------------+
                   |
          CTRL-]   |
                   |  +------------------------------------+
                   +->|void write_char(char c)             |
                      |{                                   |
                      |    putchar((int)(unsigned char)c); |
                      |}                                   |
                      +------------------------------------+
```

":tags"命令会列出现在你就已经到过哪些tag了：

ex command

```
:tags
 # TO tag          FROM line  in file/text
 1  1 write_line          8  write_block.c
 2  1 write_char          7  write_line.c
```

现在往回走. CTRL-T命令会跳到你前一次的tag处. 在上例中它会带你到调用了"write_char"的"write_line"函数的地方.

CTRL-T可以带一个命令记数，以此作为往回跳的次数，你已经向前跳过了，现在正在往回跳，我们再往前跳一次. 下面的命令可以直接跳转到当前tag序列的最后:

ex command

```
:tag
```

你也可以给它一个前缀，让它向前跳指定的步长. 比如":3tag". CTRL-T也可以带一个前缀.

这些命令可以让你向下深入一个函数调用树(使用CTRL-])，也可以回溯跳转(使用CTRL-T). 还可以随时用":tags"看你当前的跳转历史记录.

分隔窗口

":tag"命令会在当前窗口中载入包含了目标函数定义的文件[1]. 但假设你不仅要查看新的函数定义，还要同时保留当前的上下文呢? 你可以在":tag"后使用一个分隔窗口命令":split". Vim还有一个一举两得的命令:

ex command

```
:stag tagname
```

要分隔当前窗口并跳转到光标下的tag:

normal mode command

```
CTRL-W ]
```

如果同时还指定了一个命令记数，它会被当作新开窗口的行高.

多个tags文件

如果你的源文件位于多个目录下，你可以为每个目录都建一个tags文件. Vim会在使用某个目录下的tags文件进行跳转时只在那个目录下跳转[2].

[1] 译注: 如果它是在另外的文件中的话
[2]译注: 比如多个目录下的源程序中含有相同名字的标识符时

要使用更多tags文件，可以通过改变'tags'选项的设置来引入更多的tags文件．如：

ex command
```
:set tags=./tags,./../tags,./*/tags
```

这样的设置使Vim可以使用当前目录下的tags文件，上一级目录下的tags文件，以及当前目录下所有层级的子目录下的tags文件．

这样可能会引入很多的tags文件，但还有可能不敷其用． 比如说你正在编辑"~/proj/src"下的一个文件，但又想使用"~/proj/sub/tags"作为tags文件．对这种Vim情况提供了一种深度搜索目录的形式． 如下：

ex command
```
:set tags=~/proj/**/tags
```

单个tags文件

Vim在搜索众多的tags文件时，你可能会听到你的硬盘在咔嗒咔嗒拼命地叫． 显然这会降低速度．如果这样还不如花点时间生成一个大一点的tags文件．这可能要花一个通宵[1]．

这需要一个功能丰富的ctags程序，比如上面提到的那个．它有一个参数可以搜索整个目录树：

shell command
```
cd ~/proj
ctags -R .
```

用一个功能更强的ctags的好处是它能处理多种类型的文件．不光是C和C++源程序，也能对付Eiffel或者是Vim脚本． 你可以参考ctags程序的文件调整自己的需要．

现在你只要告诉Vim你那一个tags文件在哪就行了：

ex command
```
:set tags=~/proj/tags
```

同名tag

当一个函数被多次重载(或者几个类里都定义了一些同名的函数)，":tag"命令会跳转到第一个符合条件的． 如果当前文件中就有一个匹配的，那又会优先使用它．

当然还得有办法跳转到其它符合条件的tag去：

[1]译注： 没那么恐怖了，现在的PC个个威力十足，象Linux内核或Xwindows也是数分钟内可以搞定的

ex command
```
:tnext
```

重复使用这个命令可以发现其余的同名tag. 如果实在太多，还可以用下面的命令从中直接选取一个:

ex command
```
:tselect tagname
```

Vim会提供给你一个选择列表:

Display
```
  # pri kind tag               file
  1 F    f    mch_init          os_amiga.c
               mch_init()
  2 F    f    mch_init          os_mac.c
               mch_init()
  3 F    f    mch_init          os_msdos.c
               mch_init(void)
  4 F    f    mch_init          os_riscos.c
               mch_init()
Enter nr of choice (<CR> to abort):
```

现在你只需键入相应的数字(位于第一栏的). 其它栏中的信息是为了帮你作出决策的.

在多个匹配的tag之间移动，可以使用下面这些命令:

ex command
```
:tfirst                go to first match
:[count]tprevious      go to [count] previous match
:[count]tnext          go to [count] next match
:tlast                 go to last match
```

如果没有指定[count]，默认是1.

tag的名字...

命令补齐真是避免键入一个长tag名的好办法. 只要输入开头的几个字符然后按下制表符:

ex command
```
:tag write_<Tab>
```

Vim会为你补全第一个符合的tag名. 如果还不合你意, 接着按制表符直到找到你要的.

有时候你只记得一个tag名的片段. 或者有几个tag开头相同. 这里你可以用一个模式匹配来告诉Vim你要找的tag.

假设你想跳转到一个包含"block"的tag. 首先键入命令[1]:

ex command
```
:tag /block
```

现在使用命令补齐: 按<Tab>. Vim会找到所有包含"block"的tag并先提供给你第一个符合的.

"/"告诉Vim下面的名字不是一五一十的tag名, 而是一个搜索模式. 通常的搜索技巧都可以用在这里. 比如你有一个tag以"write_"开始:

ex command
```
:tselect /^write_
```

"^"表示这个tag以"write_"开始. 不然在半中间出现write的tag也会被搜索到. 同样"$"可以用于告诉Vim要查找的tag如何结束.

tags的浏览器

CTRL-]可以直接跳转到以当前光标下的word为tag名的地方去, 所以可以在一个tag列表中使用它. 下面是一个例子.

首先建立一个标识符的列表(这需要一个好的ctags):

shell command
```
ctags --c-types=f -f functions *.c
```

现在直接启动Vim, 以一个垂直分隔窗口的编辑命令打开生成的文件

shell command
```
vim
:vsplit functions
```

这个窗口中包含所有函数名的列表. 可能会有很多内容, 但是你可以暂时忽略它. 用一个":setlocal ts=99"命令清理一下显示.

在该窗口中, 定义这样的一个映射:

[1]译注: 不要急着按回车键,看下面

ex command
```
:nnoremap <buffer> <CR> 0ye<C-W>w:tag <C-R>"<CR>
```

现在把光标移到你想要查看其定义的函数名上，按下回车键，Vim就会在另一个窗口中打开相应的文件并定位到到该函数的定义上.

其它相关主题

设置'ignorecase'也可以让tag名的处理忽略掉大小写.

'tagsearch'选项告诉Vim当前参考的tags文件是否是排序过的. 默认情况假设该文件是排序过的，这会使tag的搜索快一些，但如果tag文件实际上没有排序就会在搜索时漏掉一些tag.

'taglength'告诉Vim一个tag名字中有效部分的字符个数.
================译者注---开始================

code
```
#include <stdio.h>
int very_long_variable_1;
int very_long_variable_2;
int very_long_variable_3;
int very_long_variable_4;

int main()
{
    very_long_variable_4 = very_long_variable_1 *
        very_long_variable_2;
}
```

对于上面这段代码，4个变量长度都为20，如果将'taglength'设为10，则

ex command
```
    :tag very_long_variable_4
```

会匹配到4个tag，而不是1个，光标停留在very_long_variable_1所在行上，因为被搜索的tag部分只有前面的10个字符："very_long_"，相应的显示是(是gvim中文版的真正显示，不是翻译的)：

Display
```
找到 tag: 1/4 或更多
```

================译者注---结束================

如果你使用SNiFF+，你可以使用Vim与它通讯. 参考|sniff|. SNiFF是一个商业程序.

Cscope是一个可自由使用的程序. 它不仅可以找到标识符在哪里定义, 也可以找出它们在哪里被使用. 参考|cscope|.

29.2 预览窗口

当你在写一个函数调用时, 往往需要获知这个函数的参数列表--查看函数的定义. tag所提供的机制正为此用. 最好是函数的定义可以显示在另一窗口中以免影响当前的编辑. 这可以由预览窗口提供.

要打开一个预览窗口显示函数"write_char"的定义使用命令:

ex command
```
:ptag write_char
```

Vim会打开一个窗口, 并将找到的"write_char"函数的定义显示其中. 然后光标仍保持在你的当前编辑位置. 以便你无需以CTRL-W k窗口中切换即可继续工作.

如果一个函数名出现在当前文本中, 你还可以以下面的命令在预览窗口中打开它:

normal mode command
```
CTRL-W }
```

Vim的发行包中还有一个脚本用来自动显示以当前word为tag的定义. 参考|CursorHold-example|.

要关闭该预览窗口, 使用命令:

ex command
```
:pclose
```

要在预览窗口中编辑一个文件, 使用":pedit". 这主要用于在预览窗口中编辑一个头文件, 比如:

ex command
```
:pedit defs.h
```

最后, ":psearch"可以搜索当前文件以及它所include的文件并显示匹配的行. 尤其是在使用以库的形式提供的函数时, 这时不会有对应的tags文件. 比如:

ex command
```
:psearch popen
```

这会在预览窗口中显示"stdio.h"文件，定义在popen()的原型上：

Display
```
FILE    *popen __P((const char *, const char *));
```

'previewheight'选项用于指定打开一个预览窗口时它占多少行的屏幕高度.

29.3　在程序中移动

因为程序一般是结构良好的文本，所以Vim得以识别其中的元素. 有一些特殊的命令就利用这一点提供一些便利的功能方便在程序中移动.

C程序经常有如下的结构：

code
```
#ifdef USE_POPEN
    fd = popen("ls", "r")
#else
    fd = fopen("tmp", "w")
#endif
```

但实际情况往往比这段代码长，而且很可能含有此类结构的嵌套. 把光标置于"#ifdef"上按下%. Vim就会跳转到"#else".再按下%又会跳转到"#endif". 再按下%又会回到"#ifdef"上.

当上述结构被嵌套使用时，Vim会准确找到匹配的对应元素. 这是检查是否遗漏"#endif"的好办法.

当你在"#if"-"#endif"结构的中间某个地方时，可以使用这个命令跳转到此结构的开始元素[1]：

normal mode command
```
[#
```

如果你不是在"#if"或"#ifdef"的后面使用这个命令，Vim会发出蜂鸣声以示警告. 要向前跳转到下一个"#else"或"#endif"使用命令：

normal mode command
```
]#
```

这两个命令会跳过"#if"-"#endif"块的内容. 如下例：

[1]译注： 指"#if"

code
```
#if defined(HAS_INC_H)
    a = a + inc();
# ifdef USE_THEME
    a += 3;
# endif
    set_width(a);
```

置光标于最后一行按下"[#"命令会移动到第一行． 中间"#ifdef"-"endif"块都会被跳过去．

在代码块中移动

在C代码中代码块以{}括起来． 代码块可长可短． 要移动到一个代码块的开头，使用"[["命令． "]["可以到它的末尾，这两个命令都假设"{"和"}"字符位于第一列．

"[{"命令跳转到当前代码块的开头，它会跳过与它平级的代码块．相应地"]}"会跳转到当前代码块的末尾．

来看一段例子：

code
```
                function(int a)
   +->          {
   |                if (a)
   |       +->      {
[[ |       |            for (;;)               --+
   |       |      +->   {                        |
   |    [{ |      |         foo(32);             |     --+
   |       |   [{ |         if (bar(a))  --+     | ]}    |
   +--     |      +--           break;     | ]}  |       |
           |            }              <-+       |       | ][
           +--          foobar(a)                |       |
                    }                          <-+       |
                }                                      <-+
```

写C++或Java的源代码时[1]，外层的{}用于类的定义．下一级的{}才是函数定义．光标在类中时使用"[m"可以找到前一个函数的开始． "]m"到下一个函数的末尾[2]．

另外，"[]"是向后查找一个函数的结尾，"]]"向前查找一个函数的开始，函数的结尾以第一行出现"}"为标志．

[1]译注： 推而广之，类似的语法形式的面向对象语言，如C#，Eiffel也都一样

[2]译注： 对于Java或C#，其类的成员函数的定义既是声明，正是这里的情况，对于C++，成员函数的定义可以在声明时同时指定，也可以在完成类定义后定义成员函数，这时等同于前面类似C程序的处理

code
```
                                int func1(void)
                                {
                                        return 1;
                  +----------> }
                  |
              []  |             int func2(void)
                  |       +-> {
                  |   [[  |             if (flag)
      start       +--     +--                   return flag;
                  |   ][  |             return 2;
                  |       +-> }
              ]]  |
                  |             int func3(void)
                  +----------> {
                                        return 3;
                                }
```

别忘了你还可以使用"%"在()，{}和[]之间移动，这些符号跨越多行时该命令仍然有效.

括号内的移动

"[("和"])"命令类似于"[和"]""，只不过它工作于()的内部而不是{}.

code
```
                    [(
            <---------------------------------
                      <--------
        if (a == b && (c == d || (e > f)) && x > y)
                          -------------->
                --------------------------------> >
                              ])
```

注释内的移动

向后移动到注释的开头用使用"[/"． 向前移动到其末尾用"]/"． 这只对/* - */形式的注释有效.

code
```
          +->      +-> /*
          |     [/ |    * A comment about      --+
       [/ |        +--  * wonderful life.        | ]/
          |             */                     <-+
          |
          +--           foo = bar * 3;         --+
                                                 | ]/
                        /* a short comment */  <-+
```

29.4 查找全局标识符

你在写C程序时可能经常会想知道一个变量是被声明为"int"还是"unsigned". 解决这个问题的快速办法是使用"[I"命令.

假设你的光标置身于"column"上. 键入:

normal mode command
```
[I
```

Vim会列出所有包含该标识符的行. 不光在当前文件中, 也查找当前文件所include的文件[1], 以及被include的文件所include的文件, 如此类推. 结果类似于:

Display
```
structs.h
 1:   29     unsigned     column;    /* column number */
```

使用tags或预览窗口的一个好处是include文件也会被搜索. 多数情况下你都会找到要找的东西. 即使你的tags文件已经过期, 或者你并没有头文件对应的tags.

不过, "[I"的正常工作还是需要你预先告诉Vim一些事情, 首先'include'选项要提供在何处查找include文件的信息. 它的默认值是为C和C++而设的, 对其它语言还需要做些调整.

定位include文件

Vim会在由选项'path'指定的路径里查找include文件. 如果漏掉了某个路径, 可能就找不到一些include文件了. 下面的命令可以用于检查路径是否正确:

[1]译注: 指C/C++程序中用#include语句引入的文件, 通常是头文件如stdio.h

ex command
```
:checkpath
```

它会列出所有能找到和不能找到的include文件. 输出的示例如下:

Display
```
--- Included files not found in path ---
<io.h>
vim.h -->
  <functions.h>
  <clib/exec_protos.h>
```

"io.h"文件被当前文件指定但找不到. "vim.h"可以找到, 所以":checkpath"命令继而深入"vim.h"并检查它所include的文件. 其中的"functions.h"和"clib/exec_protos.
件又没找到.

> 备注: Vim不是编译器. 它并不能识别"#ifdef"语句. 所以每个"#include"语句都会被检查, 即使它是在"#if NEVER"之后[a]
>
> [a]译注: 这里的意思是在实际编译过程该条件永远不会满足

要处理这些找不到的文件, 可以向'path'选项中再添加一个搜索路径. 到Makefile里去找要添加的路径是一个好办法, 找到那些包含了"-I"的行,比如"-I/usr/local/X11". 要把这个目录添加进去使用命令:

ex command
```
:set path+=/usr/local/X11
```

如果有多个子目录, 还可以使用通配符"*". 如:

ex command
```
:set path+=/usr/*/include
```

这会同时在"/usr/local/include"和"/usr/X11/include"下搜索文件.

所从事的项目包含一个复杂的嵌套目录结构和众多头文件时, "**"显得特别有用. 它会指示Vim遍历向下的所有子目录. 如:

ex command
```
:set path+=/projects/invent/**/include
```

这将会在下面目录中搜索include文件:

```
List
/projects/invent/include
/projects/invent/main/include
/projects/invent/main/os/include
etc.
```

其它更多细节请参考'path'.

如果你想知道实际找到了哪些include文件，使用命令:

```
ex command
:checkpath!
```

你可以会因此得到一个很长的文件列表，include文件，以及每个include文件所include 的文件，如此类推. 为缩减这个列表，Vim会显示对已经找到过的文件以"(Already listed)" 显示，并不再重复列出它们所包含的include文件了.

跳转到匹配的目标

"[I"生成的列表只含有一行上下文信息. 要仔细查看它找到的第一个匹配项，可以使用

```
normal mode command
[<Tab>
```

跳转到该项，也可以用"[CTRL-I"，CTRL-I等同于<Tab>.

"[I"命令列出的列表中每一行都有一个标号. 你要跳转到其它项时只要先键入对应的标号:

```
normal mode command
3[<Tab>
```

这会跳转到列表中的第3项. 记住CTRL-O可以把你再带回来.

相关命令

```
List
[i          只列出第一个匹配的
]I          只列出当前光标之后的匹配项
]i          只列出当前光标之后的第一个匹配项
```

查找定义的标识符

"[I"命令查找任何的标识符. 要只查找以"#define"定义的宏使用:

```
normal mode command
[D
```

同样它会遍历进include文件. 'define'选项定义了哪些行是"[D"命令所要检查的. 如果使用C和C++之外的语句你可以把它改为其它的值.

与"[D"相关的命令是:

List

[d	只列出第一个匹配
]D	只列出当前光标之后的匹配
]d	只列出当前光标之后的第一个匹配.

29.5 查找局部标识符

"[I"命令会搜索include文件. 要使搜索限制在当前文件里并执行同样的功能, 以命令:

```
normal mode command
gD
```

替代之.

提示: goto Definition. 这对查找一些静态的变量或函数很有用处. 例如(假设光标在"counter"上):

```
code
        +->   static int counter = 0;
        |
        |     int get_counter(void)
     gD |     {
        |         ++counter;
        +--       return counter;
              }
```

要进一步限制查找的范围, 让它只在当前函数里查找, 使用命令:

```
normal mode command
gd
```

它会先往回找到当前函数的开始然后向下查找当前光标下的word的第一次出现. 实际上, 它只是往回查找一个第一列是"{"字符的行. 然后再从那个位置向前查找目标标识符. 如例(假设当前光标在"idx"上):

code
```
            int find_entry(char *name)
            {
       +->      int idx;
       |
    gd |        for (idx = 0; idx < table_len; ++idx)
       |            if (strcmp(table[idx].name, name) == 0)
       +--              return idx;
            }
```

下一章: |usr_30.txt| 编辑程序
版　权: 请参考|manual-copyright| vim:tw=78:ts=8:ft=help:norl:

usr_30.txt Vim 6.3版.最后修改: 2004年01月17日

VIM用户手册--- 作者: Bram Moolenaar

程序的编辑

Vim有一些命令专用于辅助编码. 比如直接在编辑器里编译源程序, 跳转到编译器报错的行号上. 或者自动为众多语言设置相应的缩进, 以及对注释的格式化等等.

下一章: |usr_31.txt| 探索GUI的世界
前一章: |usr_29.txt| 在源程序中移动
目 录: |usr_toc.txt|

30.1 编译

Vim有一个叫"quickfix"的命令集. 这些命令可以让程序员在Vim中编辑程序, 如果编译器报告了程序的错误, 你还能在这些错误中自由遍历, 修补后再重新编译, 如此反复, 直到你的程序编译无误为止.

下面的命令运行程序"make"(如果你要跟什么参数的话, 直接放在后面就行了)并捕捉它可能的错误:

ex command
```
:make {arguments}
```

如果有错误产生, Vim会自动将光标定位于第一个错误的事发地点.

看一下下面这个使用":make"的例子. (典型情况下, :make命令会产生众多的错误, 其中还有一些看起来很蠢错误) ":make"命令执行之后的屏幕显示可能是这样的:

Display
```
:!make | &tee /tmp/vim215953.err
gcc -g -Wall -o prog main.c sub.c
main.c: In function 'main':
main.c:6: too many arguments to function 'do_sub'
main.c: At top level:
main.c:10: parse error before '}'
make: *** [prog] Error 1

2 returned
"main.c" 11L, 111C
(3 of 6): too many arguments to function 'do_sub'
Hit ENTER or type command to continue
```

从这里你可以看出文件"main.c"中有错误. 按下<Enter>之后, Vim会载入文件"main.c"并把光标定位在第6行, 也就是引发第一个错误的程序行. 整个过程不需要你来指示哪个文件哪一行, Vim会根据错误信息自动判断事发现场.

code
```
          +--------------------------------------------------+
          |int main()                                        |
          |{                                                 |
          |       int i=3;                                   |
cursor -> |       do_sub("foo");                             |
          |       ++i;                                       |
          |       return (0);                                |
          |}                                                 |
          |}                                                 |
          | ~                                                |
          |(3 of 12): too many arguments to function 'do_sub'|
          +--------------------------------------------------+
```

下面的命令会带你到下一个错误的事发地点:

ex command
```
:cnext
```

此例中Vim会跳转到文件的第10行, 那里多出来了一个"}"字符.

屏幕上可用的空间紧张时, Vim会缩减必需显示的错误信息.要查看完整的错误信息描述, 使用下面的命令:

ex command
```
:cc
```

":clist"命令会列出一个所有错误信息的列表，输出的信息类似于：

Display
```
:clist
3 main.c: 6:too many arguments to function 'do_sub'
5 main.c: 10:parse error before '}'
```

这里只列出了包含文件名和一个行号的简要信息，Vim认为这些对于理解一个错误是最重要的，其它的行相对就没那么要紧了． 然而事实上有时候被忽略掉的那些行正是你想要的东西．比如链接器报告说哪个函数没有定义．这只要在":clist"命令后加一个"!"就可以列出完整的错误信息列表：

Display
```
:clist!
1 gcc -g -Wall -o prog main.c sub.c
2 main.c: In function 'main':
3 main.c:6: too many arguments to function 'do_sub'
4 main.c: At top level:
5 main.c:10: parse error before '}'
6 make: *** [prog] Error 1
```

Vim会以特殊颜色显示当前的错误．要回到前一个错误去，使用命令：

ex command
```
:cprevious
```

此外还有一些其它的命令用于遍历整个错误列表：

List

:cfirst	到第一个错误处
:clast	到最后一个错误处
:cc 3	到第3个错误处

使用其它的编译器

'makeprg'选项定义了":make"命令被执行时要实际调用的外部程序．通常情况是"make"程序，不过Visual C++的用户就需要把它定义为"nmake"[1]：

ex command
```
:set makeprg=nmake
```

这个选项中还可以跟上命令的参数，注意其中的特殊字符要用反斜杠来特殊处理：

[1]译注： "nmake"随同Visual C++发布．是微软版的make

ex command
```
:set makeprg=nmake -f project.mak
```

在这个选项[1]中还可以引用一些Vim的关键字. 字符%代表当前的文件名. 所以如果你这样设置该选项:

ex command
```
:set makeprg=make %
```

当你编辑的文件是main.c时, ":make"命令就会实际执行:

ex command
```
make main.c
```

这在实际中用处不大, 最好是对上面的设定稍事调整, 使用:r这个修饰标志:

ex command
```
:set makeprg=make %:r.o
```

现在命令执行起来就是:

ex command
```
make main.o
```

关于这些修饰标志, 请参考: |filename-modifiers|

上一个错误列表

假设你正在用命令":make"来构建一个程序, 编译过程中出现了两个错误[2], 一个是位于某个源文件中的一个警告, 另一个是位于另一个是位于另一源文件中的错误. 一般来说你会先去修改产生了"错误"的源文件, 然后重新编译该文件以检查是否真正排除了这个错误, 但是现在你却不能在现在的错误列表中看到刚才的那个警告信息了, 因为这一次的":make"只是针对产生错误的这一个文件, 而不是整个项目[3], 还好, Vim在内部维护了一个关于错误列表的列表.每次的":make"命令都会产生一个错误列表, 多次执行":make"就形成一个关于错误列表的列表, 你可以用下面的命令回到前一个错误列表:

ex command
```
:colder
```

[1]译注: 规律:Vim的命令行上允许使用一些特殊字符
[2]译注: 此处的错误兼指Warning和Error
[3]译注: 一般来说, 编译器将编译过程中产生的问题按严重程度分门别类,比如warning, error, fatal error. warning的严重程度较之error为轻, fatal error最严重, 所以此处举例中先去解决error

现在你就可以使用":clist"命令和":cc {nr}"命令来针对前一个错误清单进行操作，回到刚才引起warning的位置了.

要向前查看下一个错误清单，使用下面的命令：

ex command
```
:cnewer
```

Vim可以同时记住10个错误清单列表.

使用不同的编译器

如此诱人的功能不是白给的，你得告诉Vim你所用的编译器产生的错误信息的格式是什么样子. 这可以通过对'errorformat'选项的设置来完成. 该选项的语法略有些复杂不过也正因为如此它才可以应用于几乎是任何的编译器. 你可以在|errorformat|处发现对该选项的解释.

有可能你在工作中要使用不同的编译器.不过每次都去分别设置'makeprg'和'errorformat'可不是一件容易的事. Vim为此提供了一个简易办法，比如说，现在你要切换到使用Microsoft Visual C++的编译器就可以使用下面的命令：

ex command
```
:compiler msvc
```

这个命令会自动查看关于"msvc"这个编译器设置的Vim脚本并应用里面的设置. 你也可以自己量身打造这样一个为某种编译器进行设置的脚本文件. 请查看|write-compile-plugin|.

输出重定向

":make"命令的幕后工作是把编译过程的输出重定向到一个记录错误信息的文件中. 这个过程需要几个Vim选项的紧密配合，比如'shell'. 如果你的":make"命令不能捕获错误,请检查'makeef'和'shellpipe'这两个选项，'shellquote'和'shellxquote'的设置也可能与此有关.

如果你不能把":make"命令重定向到文件，还有别的办法，你还可以单独编译程序并把编译输出重定向到一个文件中，然后使用下面的命令读取这个错误信息文件：

ex command
```
:cfile {filename}
```

这会跟使用":make"命令一样能让你跳转到出错地点去.

30.2 C程序的缩进

程序缩进得当的话会大大提高可读性. Vim有几种措施来协助进行缩进.

对C程序来说打开'cindent'选项即可, Vim对C程序有良好的支持, 它会为你的C程序以良好的风格缩进做大量的工作. 对于多级缩进你还可以通过'shiftwidth'选项的值来调整缩进量. 4个空格是一个不错的选择. 也可以用一个":set"命令备齐这两个选项:

ex command
```
:set cindent shiftwidth=4
```

在上面的设置下, 你若键入了"if (x)"这样的语句, 那么下一行就会自动向右缩进一级.

List
```
                                if (flag)
自动缩进            --->            do_the_work();
自动减小缩进 <--                if (other_flag) {
自动缩进            --->            do_file();
保持当前缩进                        do_some_more();
自动减小缩进 <--                }
```

如果你是在花括号里键入一个语句块, Vim会在语句块的开始进行缩进, 在语句块以"}" 结束时减小缩进, 因为毕竟Vim无法得知你会在中间写下什么东西.

使用自动缩进的另一个辅助作用是帮助你发现程序里的错误. 当你键入一个}符号来结束一个函数的定义时, 如果发现缩量与你的期望有出入时, 很可能是在函数体中哪里漏掉了一个右花括号}. 可以使用"%"命令来帮你找到当前的右花括号跟哪一个左花括号相匹配.

丢了)和;符号也一样会引起异常的缩进. 所以如果你看到缩进量多于预期时, 最好检查一下前面的程序段.

如果你正在接手一些缩进格式相当糟糕的代码, 或者要在已有的代码里增删修补时. 或许你会想重新对这些代码的缩进进行整理. 使用"="操作符命令, 最简单的形式是:

normal mode command
```
==
```

这个简单的命令会重新为当前行进行适当的缩进. 与其它的操作符命令一样, 它有三种命令形式. 在Visual模式下"="命令为被选择的行设定缩进. 对于可用的文件对象, "a{" 是有最用的--它选择的对象是当前的{}程序块. 所以下面的命令会为当前的代码块重新设定缩进:

normal mode command
```
=a{
```

如果手头的代码实在是糟糕透顶，你也可以用下面的命令重新为整个文件进行缩进：

normal mode command
```
gg=G
```

不过，对于用手工精心打造出来的缩进格式可不要轻易这么做，虽然Vim的自动缩进功能不错，不过人们总是有自己的偏好.

设置缩进风格

不同的人喜欢不同风格的缩进，Vim对缩进风格的设置对90%的程序来说正是他们所喜欢的．不过，还是应该允许不同的风格存在，你可以在'cinoptions'选项里定义自己的缩进风格.

默认情况下'cinoptions'选项的值是一个空字串，Vim会使用默认的风格．如果你希望有改变某些情况下的缩进风格．比如，让花括号的缩进看起来象下面这样：

code
```
if (flag)
  {
    i = 8;
    j = 0;
  }
```

可以使用下面的命令：

ex command
```
:set cinoptions+={2
```

还可以设置很多这样的在特定情形下的缩进风格，请参考|cinoptions-value|.

30.3　自动缩进

虽然由编辑器为你的程序进行缩进是个不错的主意，你也不会想每次打开一个C文件时都去手工把'cindent'选项打开，这样的工作完全可以自动化：

ex command
```
:filetype indent on
```

实际上，这个命令所做的远不至仅仅为C文件打开'cindent'选项这么简单．首先，它会检查文件的类型．这与设置语法高亮时所做的类型检查是一样的.

一旦Vim确定了文件类型，它就会为此类型的文件搜索一个对应的定义其缩进风格的文件. Vim的发布版中包含了很多为每种语言设置不同缩进风格的脚本文件. 这些脚本文件正是为你在编辑中使用自动缩进功能进行准备工作.

如果你不想用自动缩进的话，还可以再把它关闭掉:

ex command
```
:filetype indent off
```

如果你只是不想对某种类型的文件使用自动缩进，可以创建一个只包含下面一行的文件:

ex command
```
:let b:did_indent = 1
```

现在给它起个名字保存为:

List
```
{directory}/indent/{filetype}.vim
```

其中的{filetype}就是你想避免它自动缩进的文件类型，比如"cpp"或"java". 你可以用下面的命令查看Vim检测到的当前文件类型到底是什么:

ex command
```
:set filetype
```

对现在这个文件来说，它的类型是:

Display
```
filetype=help
```

这里你的{filetype}就是"help".

对于{directory}部分你需要知道你的运行时目录. 检查下面命令的输出:

ex command
```
set runtimepath
```

使用第一个逗号之前的那一项即可. 如果它的输出看起来是:

Display
```
runtimepath=~/.vim,/usr/local/share/vim/vim60/runtime,~/.vim/after
```

你的{directory}就可以指定为"~/.vim". 所以完整的文件名就是:

List
```
~/.vim/indent/help.vim
```

除了简单地关闭自动缩进功能之外，你还可以定制自己的自动缩进文件. 在|indenting-expression|里对此有详细解释.

30.4 其它的种种缩进

使用自动缩进最简单的形式是打开'autoindent'选项. 它使用当前行前面一行的缩进.一个更智能一点的方案是使用'smartindent'选项. 这主要用于没有缩进文件的程序语言. 'smartindent'并没有'cindent'对缩进考虑的那么周全，不过它比'autoindent'还是智能一点.

打开'smartindent'选项的话，每一个开花括号之后都会增加一级的缩进，而在对应的闭花括号}之后再撤去这一级的缩进. 此外，也会在遇到'cinwords'选项中列出的词时增加一级缩进. 以符号#开始的行也会被特殊对待：不使用任何缩进. 这样所有的预处理器指示器就都从第1列开始[1]. 下一行又会恢复在此之前的缩进量

修正缩进

如果你使用'autoindent'或'smartindent'得到了前一行的缩进量，很可能这个缩进量不是你刚好想要的. 一个快速增减当前缩进量的办法是在Insert模式下使用CTRL-D[2]和CTRL-T[3].

比如，你正在键入如下的shell脚本：

shell command
```
if test -n a; then
   echo a
   echo "--------"
fi
```

在开始之前你设置了如下的选项：

ex command
```
:set autoindent shiftwidth=3
```

键入第一行之后，你按下回车键开始第二行：

code
```
if test -n a; then
echo
```

[1]译注： 这里作者的假设是C/C++语言
[2]译注： 减小一个缩进单位
[3]译注： 增加一个缩进单位

现在你需要增加该行的缩进量. 按CTRL-T. 结果将是:

code
```
        if test -n a; then
           echo
```

Insert模式下的CTRL-T命令会向当前的缩进量附加由'shiftwidth'指定的字符宽度. 不管光标的当前位置在该行中的任何位置都可使用这一命令[1].

继续刚才第二行, 按下回车开始第三行, 第三行的缩进刚好, 但是再回车之后的第四行看起来就...:

code
```
        if test -n a; then
           echo a
           echo "--------"
           fi
```

按CTRL-D可以移除第4行中超额的缩进量. 它所减小的缩进量也是'shiftwidth'指定的字符宽度, 同样, 这一命令不管光标位于该行的何处都一样发挥作用.

在Normal模式下, 你也可以使用">>"和"<<" 命令来向右/左[2]. ">"和"<"其实是操作符命令, 所以仍然可以使用此类命令的3 种形式[3]. 下面是一个比较有用的组合:

normal mode command
```
        >i{
```

这为{}内的所有行增加一个缩进单位[4]. ">a{"命令则包括了{ 与}所在的行本身. 假设下例中的光标停于"printf"上[5]:

Display

原始内容	">i{"命令之后	">a{"命令之后
if (flag)	if (flag)	if (flag)
{	{	{
printf("yes");	printf("yes");	printf("yes");
flag = 0;	flag = 0;	flag = 0;
}	}	}

[1]译注: 来自使用<Tab>的经验使多数人直觉光标应处在第一列才会把后面的所有内容向后"推", 不是这样的!

[2]译注: 规律:Vim一些命令具有象形文字的示意功能

[3]译注: 规律:Vim的命令如此众多的一个重要原因就是这种基本命令*操作模式产生的爆炸组合

[4]译注: 这里的一个缩进单位即是指由'shiftwidth'选项指定的字符宽度

[5]译注: ">a{"命令之后是指以原始内容为基础执行该命令, 而不是基于第一个">i{"命令执行后的结果

30.5 制表符和空格

'tabstop'选项的默认值是8． 虽然你可以改变这个值，不过很快你就会遇到麻烦．因为其它的程序都假设一个制表符占据8个字符宽度，似乎突然之间你的文件看起来就大不一样了． 另外，大多数打印机也都假设一个固定制表符的宽度是8． 所以最好还是保留'tabstop'的值为8不要动．(如果你在编辑一个制表符宽度不是8的文件，请参考|25.3|里的建议)．

但对于源程序的缩进来说，如果使用8个字符的话，程序行很容易就会超屏幕的可视区． 而用1个字符宽度的话又太不显眼． 所以很多人选择了4，这是一个不错的折衷．

因为一个制表符是8个字符而你想用4个空格来进行缩进，所以不能通过插入制表符来进行缩进． 有两个办法可以解决这个问题：

1. 混合使用制表符和空格．因为一个制表符占8个字符的宽度，所以你的文件就可以少些字符．而插入一个制表符也比插入8个空格快得多，使用退格键也挺快．

2. 只用空格．这可以避免有些程序使用不同的制表符宽度引起的问题．

幸运的是Vim能同时支持上面两种办法．

空格和制表符

如果你是在混合使用空格和制表符，你就只管编辑吧，Vim会在幕后处理这些工作．

你可以使用'softtabstop'选项来简化工作．这个选项告诉Vim让一个<Tab>看起来就象是由'softtabstop'所设置的宽度，但实际上是对制表符和空格的混合．

执行下面的命令后，每次你按下制表符键，光标就会向前移动4列：

ex command
```
:set softtabstop=4
```

当你在第一列按下制表符后，文件中被插入4个空格．再按下制表符Vim会在4个空格后加一个制表符(这样一共是8列)． Vim以此来尽可能地使用制表符，不足的宽度再以空格来补充．

使用退格键时是另一种情况． 一个退格键总是删除由'softtabstop'指定的宽度．同样剩余的空白会尽可能多地使用制表符，留下的间隙用空格来填充．

下例图示了按下跳格键及使用退格键时的情景． 一个"．"代表一个空格，"------->"代表制表符．

Display

输入	结果
<Tab>	....
<Tab><Tab>	------->
<Tab><Tab><Tab>	------->....
<Tab><Tab><Tab><BS>	------->
<Tab><Tab><Tab><BS><BS>	....

另一个办法是使用'smarttab'选项．打开该选项时，Vim会在缩进时应用'shiftwidth'选项的值[1]，而在第一个非空白字符之后再按下<Tab>就插入真正的制表符．不过此时的退格键就不象使用'softtabstop'时那么配合了．

只用空格

如果你根本不想在文件中看到制表符，你可以设置'expandtab'选项：

ex command
```
:set expandtab
```

打开该选项之后[2]，按下制表符就会插入相应宽度的空格来代替．看起来空白区域的宽度都是一样，但是文件中实际上不会有制表符．

此时的退格键只会删除一个空格，所以按下一个<Tab>键你得按8次退格键才能删除由它所插入的空格．如果这些空白是缩进，按下CTRL-D 会更快．

把制表符变为空格(或反之)

打开'expandtab'选项并不会影响到此前输入的制表符．换句话说，文件里原来有多少制表符还在还是不变，如果你想把所有的制表符都转为空格，可以使用":retab"命令：

ex command
```
:set expandtab
:%retab
```

现在Vim就会把所有缩进中的制表符改为空格了． 但是在非空白字符之后的制表符还是保留了下来． 如果你想把所有的制表符都转为空白，在命令后加一个[3]!：

[1]译注： 意为一级缩进的宽度是由'shiftwidth'，具体插入空格还是制表符还得看当前缩进量是否够一个tabstop
[2]译注： 用语:打开某选项暗示这个选项的值类型是boolean型，只有开和关两种状态
[3]译注： !加在命令后表示强制，全部，忽略

ex command
```
:%retab!
```

这样作有一些风险，因为这可能会改变那些作为字符串内容的制表符．你可以用下面的搜索命令检查当前文件里有没有这样的情况：

normal mode command
```
/"[^"t]*t[^"]*"
```

不过最好还是不要在字符串中以一个实实在在的<Tab>来代表制表符，可以用"\t"来代替它以避免此类的麻烦．

要反过来转换这样即可：

ex command
```
:set noexpandtab
:%retab!
```

30.6 对注释的格式化

Vim的另一个引人之处是它能理解注释． 你可以让Vim来格式化你的注释． 假如你有下面的注释：

code
```
/*
 * This is a test
 * of the text formatting.
 */
```

你可以把光标置到注释的开始，然后用如下命令来格式它：

normal mode command
```
gq]/
```

"gq"是格式化命令的操作符．"]/"是移动到注释结尾的移动命令．执行后结果是：

code
```
/*
 * This is a test of the text formatting.
 */
```

注意Vim已经正确处理了注释中的每一行的开头．

另一个办法是在Visual模式下先选定要格式化的内容然后再按下命令"gq".

要在注释中加入一行，把光标放在其中一行上，然后按"o". 结果看起来会是:

code
```
/*
 * This is a test of the text formatting.
 *
 */
```

Vim会自动为你新加的注释行插入一个星号和一个空格. 现在你直接写注释就行了. 如果你写入的内容超出了'textwidth'的限制，Vim还会自动为你断行，断行时它同时不会忘了为新行注释插入星号和一个空格:

code
```
/*
 * This is a test of the text formatting.
 * Typing a lot of text here will make Vim
 * break
 */
```

要达到上述功能就必需在'formatoptions'选项里设置正确的标志:

List

r	在Insert模式下按下回车时插入一个星号
o	在Normal模式下按"o"或"O"时插入一个星号
c	根据'textwidth'的设置自动为注释断行

请参考|fo-table|了解更多的标志字符.

自定义注释格式

'comments'选项定义一个注释的外观. Vim会区别对待单行注释和那种对头，体，尾各有规定的多行注释.

很多的多行注释都以一个特殊字符开头，C++中用//，Makefiles中用#，Vim脚本用双引号". 比如，要让Vim理解C++的注释:

ex command
```
:set comments=://
```

其中的冒号是为了分隔标志和后面的注释关键字，Vim就是根据这里的注释关键字来识别一个注释的. 'comments'选项的内容一般形式是:

Display
```
{flags}:{text}
```

{flags}部分可以是空的，就象上面的例子中一样.

多个这样的注释项定义可以串连在一起，其间用逗号分隔，这样就可以同时定义多种类型的注释. 比如，若要想在回覆e-mail信息时，使它人写的以">"和"!"字符开头的内容成为注释，可以这样:

ex command
```
:set comments=n:>,n:!
```

这里有两个注释项定义，分别定义了以">"开头的注释和以"!" 开头的注释. 两者的定义都使用了标志"n". 该标志是说这些注释可以是嵌套的. 这样以">"开头的后面内容本身也可能是注释，这样的定义下格式化后的内容可能是这样:

Display
```
> ! Did you see that site?
> ! It looks really great.
> I don't like it.  The
> colors are terrible.
What is the URL of that
site?
```

试着为'textwidth'选项设置不同的值，然后在Visual模式下选定这段文本以"gq"命令格式化，结果可能是[1]:

Display
```
> ! Did you see that site? It looks really great.
> I don't like it.  The colors are terrible.
What is the URL of that site?
```

注意到没有，Vim不会把属于一种类型的注释文本挪到另一种类型的注释中去. 第2行里的"I"本可以放在第1行的末尾，但因为第1行以">!"而第2行以">"开头，所以Vim认为这是两种不同类型的注释. 因此要放在不同的行上.

三段式注释

典型的C风格的多行注释以"/*"开头，中间的注释行以"*"开头，注释的最后以"*/"结尾. 这三段对应于'comments'这样的形式:

ex command
```
:set comments=s1:/*,mb:*,ex:*/
```

[1]译注: gq命令根据indent, comments, textwidth等多个选项进行工作

注释的开头以"s1:/*"定义. "s"意为start. 冒号是用于分隔标志字符和其后的注释关键字"/*". 注释中还有一个"1". 这是告诉Vim三段式注释的中间行要有一个字符宽度的右偏移.

中间部分的定义是"mb:*", 其中"m"意为middle, "b"标志是说注释关键字"*"之后要有空格. 否则Vim会认为象"*pointer"这样的指针定义也是三段式注释的一部分.

结尾的定义:"ex:*/", "e"意为注释的结尾(end)[1]"x"标志在这里有一层特殊含义, 它告诉Vim自动插入一个星号后, 如果接下来输入了一个/字符, 就要移除多余的空格.

更多的信息请参数|format-comments|.

下一章:	|usr_31.txt| 探索GUI
版　权:	请参考|manual-copyright| vim:tw=78:ts=8:ft=help:norl:

[1]译注: 指对三段式注释结束的标识

usr_31.txt Vim 6.3版.最后修改: 2003年10月21日

VIM用户手册--- 作者: Bram Moolenaar

探索GUI

Vim在它的传统阵地终端上表现良好，但是一些GUI特性更使其锦上添花. 文件浏览器可用于所有需要文件名的命令. 对话框则可方便用户进行选择. 另外，快捷键也大大加快了访问菜单的速度.

下一章: |usr_40.txt| 定义新命令
前一章: |usr_30.txt| 程序的编辑
目 录: |usr_toc.txt|

31.1 文件浏览器

使用File/Open...菜单命令你会得到一个文件浏览器，你可以在熟悉的界面中选择要编辑的文件. 但如果你想将在一个分隔窗口中打开该文件呢? 菜单里并没有这个命令. 当然你可以先用Window/Split命令再用File/Open...命令，但这毕竟太繁琐了.

既然我们已经习惯于在Vim中使用手工键入的命令，何不通过一个命令来打开一个文件对话框呢. 要让分隔命令从文件浏览器中选择文件，只需在该命令前加上"browse":

ex command
```
:browse split
```

":split"命令会另辟窗口打开选择的文件. 如果你撤消了选择文件的操作，整个命令都会被中止，窗口并不分隔.

也可以在此命令后面指定一个文件/目录名作为参数. 这使得文件浏览器以此作为被默认选取的文件/目录. 如:

ex command
```
:browse split /etc
```

文件浏览器弹出后，以"/etc"目录作为起始目录.

":browse"命令可以作为任何打开文件的命令的前缀[1].

没有指定起始目录时，Vim会自己选择一个起始目录. 默认情况下是它上次打开的目录.这样你用":browse split"命令选取"/usr/local/share"下的文件后，下一次的文件选取对话框就将以"/usr/local/share"目录作为起始目录.

'browsedir'选项可以控制Vim如何选择起始目录. 它可以取下面的几个值:

List

last	用上次访问过的目录(默认)
buffer	用当前文件所在的目录
current	用当前工作目录

假设你的工作目录是"/usr"，编辑的文件是"/usr/local/share/readme"，则下面的命令:

ex command
```
:set browsedir=buffer
:browse edit
```

将打开一个以"/usr/local/share"为起始目录的文件选取对话框. 而命令:

ex command
```
        :set browsedir=current
        :browse edit
```

则以"/usr"作为起始目录.

> 备注: 为了避免用户对鼠标的依赖，多数文件浏览器都可以用键盘进行操作. 不过各个系统上的操作方法不尽相同，这里就不做详细介绍了. Vim会尽可能使用标准的文件浏览器，你的系统文档里应该会解释它对应的键盘操作.

不用GUI版本时，你还可以通过文件浏览窗口来选择文件. 不过，这跟":browse"命令就无关了. 请参考|file-explorer|.

31.2 确认

[1]译: 类似于vertical，hide

Vim会对诸如覆盖同名文件或其它可能的损失提供保护. 如果你的行为具有明显的错误征兆. Vim会发出一个错误信息, 同时告诉你要真想这样做的话在命令后加一个!, 表明你原意后果自负. 为避免再次输入带一个!后缀的命令, 你也可以让Vim给你一个对话框进行确认. 你可以通过对"OK"或"Cancel"的选择告诉Vim你想做什么. 例如, 你对一个文件作出了一个改动. 此时转而去编辑另一个文件:

ex command
```
:confirm edit foo.txt
```

Vim会弹出一个这样的对话框:

Display
```
+-----------------------------------+
|                                   |
|   ?   Save changes to "bar.txt"?  |
|                                   |
|   YES   NO              CANCEL    |
+-----------------------------------+
```

在这里你可以进行几种选择. 如果你想保存这些改动, 先"YES". 如果不就选"NO". 如果你已经忘了刚才就做了什么想回头再看一下那就用"CANCEL". 这时你会回到原来的文件, 作出的改动仍然保留.

正如":browse"命令一样, ":confirm"命令可以作为任何编辑另一个文件的命令的前缀[1]两者还可以组合使用:

ex command
```
:confirm browse edit
```

如果当前缓冲区有所改动的话就会弹出一个确认对话框. 然后再给用户一个文件浏览器选择要编辑的文件.

> 备注: 在确认对话框中你可以用键盘进行操作. 通常情况下制表符<Tab>键和光标键都可以用来作出选择. 按下回车键作出最终的选择. 不过不同的系统还是可能有所不同.

即使不用GUI版本, ":confirm"命令也可以照常工作. 不过此时它不是弹出一个对话框. 而是在窗口底部显示一条信息, 同时等待你按下一个键来作出选择.

Display
```
:confirm edit main.c
Save changes to "Untitled"?
[Y]es, (N)o, (C)ancel:
```

[1]译: 确切说, ":confirm"命令可以作为任何命令的前缀, 不过只有在Vim需要用户确认时才弹出相应的对话框, 试一下命令":confirm echo 'asdf'" 和":confirm q"

现在你可以按下一个键来作出选择. 这不象其它在命令行上的命令, 不需再按回车. 按下对应的选择键就马上生效.

31.3 菜单命令的快捷键

通过键盘可以使用所有的Vim命令. 菜单则是一种简单一些的方法, 你无需记住命令的名字. 不过你就必需把手从键盘上移开去抓住鼠标.

不过菜单也可以通过键盘访问. 具体方法因系统而异, 但通常情况下用法是这样的: 用<Alt>键盘加上菜单中带下划线的字母. 比如<A-w>(<Alt>和w)键就可弹出Window 菜单

在Window菜单中, "split"命令中p带有下划线. 要选择该命令, 只需松开<Alt>键盘按下p.

用<Alt>键激活一个菜单后, 你可以用箭头键来访问各个菜单命令. <Right>是打开一个子菜单, <Left>则可关闭它. <Esc>也可关闭一个菜单[1]. <Enter>也可打开一个子菜单, 按下回车键最终选取菜单命令.

<Alt>键既然可以用于选择菜单, 又可以用作键盘映射的组合键. 这就存在着冲突的可能性. 'winaltkeys'选项专用于控制Vim对<Alt>键的处理.

它的默认值是"menu", 这是个不错的折衷: 如果键盘映射中包含的字符代表了某个菜单那它就不能再进行键盘映射. 其它字符则可一如既往地进行键盘映射.

将该选项设为"no"则禁用了通过<Alt>与字符键的组合进行菜单访问. 此时你就必需使用鼠标了, 键盘映射则无所不能.

取值为"yes"意为可以用任何字符跟<Alt>组合来访问菜单. 有些组合可能就不止是选取菜单而是代表了其它的操作[2].

31.4 Vim的窗口位置和大小

下面的命令可以得到当前Vim窗口的坐标位置:

ex command
```
:winpos
```

该命令只对GUI版本有效. 输出的结果大致是:

Display
```
Window position: X 272, Y 103
```

[1]译: 可关闭当前菜单, 如子菜单或主菜单
[2]译者: 如<A-F4>关闭当前Vim程序

给出的位置信息是以屏幕象素为单位的. 你可以据此控制Vim窗口的位置. 比如下面的命令可以把窗口左移100象素:

ex command
```
:winpos 172 103
```

备注: 报告出来的位置跟实际位置可能会有少许偏差, 这是由于窗口管理器处理窗口的边界造成的.

你也可以把该命令放入你的初始化脚本中控制窗口的初始位置.

Vim窗口的大小则是以字符为单位的. 所以它的实际大小还有赖于所用的字体大小. 下面的命令可以看出目前的窗口大小:

ex command
```
:set lines columns
```

通过设置'lines'和/或'columns'选项的值可以改变窗口的大小:

ex command
```
:set lines=50
:set columns=80
```

获取窗口大小对终端版本的Vim和GUI版本一样. 不过并不是任何终端都可以改变窗口的大小.

启动X-Windows上的gvim时可以在命令行上指定窗口的大小和位置:

shell command
```
gvim -geometry {width}x{height}+{x_offset}+{y_offset}
```

{width}和{height}以字符为单位, {x_offset}和{y_offset}则以屏幕象素作为度量单位. 如:

shell command
```
gvim -geometry 80x25+100+300
```

31.5 其它

你可以在gvim里写email. 你需要在你的e-mail程序里把gvim设为默认的编辑器. 不过即使这样你还是会发现这样不行: 你的mail程序会认为已经完成的编辑, 而你的gvim 还正在运行!

问题在于gvim启动后就断绝了它和shell的联系. 你从终端上启动gvim时这倒是不错, 如此一来你还可以在该终端里做其它的事. 但如果你想让当前的终端等待gvim的结束, 就必需阻止它和启动它的shell断绝关系. "-f"选项正是这个用途:

shell command
```
gvim -f file.txt
```

"-f"意为foreground(前台). 现在Vim会挂起启动它的shell直到它退出.

在Vim中启动GUI

在Unix上可以先在终端中启动Vim. 一旦你编辑哪个文件时决定要用GUI版本，可以用命令

ex command
```
:gui
```

随时进入.

Vim会丢弃当前的终端窗口转而打开一个GUI窗口. 这样你可以继续使用你的终端. "-f"参数还是可以用在这里来保持GUI作为一个前台运行的命令. 即":gui -f".

GVIM的初始化文件

gvim启动时，它会读取gvimrc这个初始化文件. 类似于启动Vim时的vimrc. gvimrc可以用来配置一些只用在GUI版本中的命令. 比如，你可以用'lines'选项来控制窗口的大小:

ex command
```
:set lines=55
```

一般来说你不会想在一个终端窗口中这样做，因为它的大小是固定的(除非象xterm那样的终端也支持调整窗口大小).

gvimrc初始化文件跟vimrc在同一目录. 通常是Unix上是"~/.gvimrc"，MS-Windows上则是"$VIM/_gvimrc".

如果因为某种原因你想使用通常的gvimrc文件之外的某个文件作为初始化文件，还可以在启动时用"-U"来指派另一个文件:

shell command
```
gvim -U thisrc ...
```

这可以让gvim以不同的面貌出现. 比如你可以设定它的窗口以不同的大小.

如果只是想避免读取来自gvimrc文件中的配置，只需用命令[1]:

[1]译：不过注意还是会读取vimrc配置文件，避免读取vimrc的是-u NONE

shell command
```
gvim -U NONE ...
```

下一章：|usr_40.txt| 定义新命令
版　权：　请参考|manual-copyright| vim:tw=78:ts=8:ft=help:norl:

usr_40.txt Vim 6.3版.最后修改：2004年02月13日

VIM用户手册--- 作者： Bram Moolenaar

定义新命令

Vim是一个可扩展的编辑器． 你可以把自己常用的命令序列定义为一个新的命令． 或者对已有的命令进行重新定义．自动执行的命令可以让命令在特定情形下被自动执行．

下一章： |usr_41.txt| Vim脚本
前一章： |usr_31.txt| 探索GUI
目 录： |usr_toc.txt|

40.1 键映射

在|05.3|节中对简单的映射已经有了一些介绍． 基本思想是把一系列击键解释为另外的按键序列． 这是一种简单而强大的机制．

最简单的形式莫过于把一个单个的键映射为一系列其它键了.因为除<F1>之外的功能键在Vim中并没有预定义内容，所以映射这些键是最理想不过了． 如:

ex command
```
:map <F2> GoDate: <Esc>:read !date<CR>kJ
```

这个例子展示了如何在映射命令中使用3种不同的模式． 在用"G"命令定位到文件尾之后，"o"命令开始一个新行进行编辑． 接下来的"Date: " 被插入到该行中，然后一个<Esc>又让你退出了Insert模式．

注意此例中<>里的特殊表示． 这叫做尖括号表示法． 用这种记法，特殊键不需要你真正按下这些键来表示，而只需在尖括号里键入他们的名字描述[1]． 这样可以使整个命令的可读性更强，同时也避免了你在copy & paste这些命令时引起一些问题．

":"字符会让Vim工作于冒号命令行模式． ":read !date"命令读取"date"命令的输出并把它追加到当前行之后． <CR>对于":read"命令的执行是必需的．

[1]译注： 但这种描述是由Vim定义的，当然不能任意来描述

命令执行到这里看起来会是这样:

Display
```
Date:
Fri Jun 15 12:54:34 CEST 2001
```

现在"kJ"将会把光标向上移一行并把两行的内容粘连为一行.

要知道已经有哪些键被映射了，请参考|map-which-keys|.

键映射和模式

":map"命令定义了在Normal模式下对键序的重新映射． 你同样可以定义工作于其它模式的键序．如":imap"可以定义Insert模式下的键序映射． 下面的键映射可以在Insert模式下在当前光标下插入当前日期[1]:

ex command
```
:imap <F2> <CR>Date: <Esc>:read !date<CR>kJ
```

它起来很象我们刚才在Normal模式下为<F2>所作的映射，只是开头不同．为Normal模式所做的<F2>映射仍然不受影响．所以可以对同一个键序在不同模式下映射不同的内容.

注意上例中，虽然该映射的工作始于Insert模式，它结束时却是在Normal模式． 如果你想在映射工作结束时仍处于Insert模式，可以在上面的映射命令最后放一个"a".

下面是一个各工作模式下对应的映射命令的列表:

List
```
:map      Normal, Visual and Operator-pending
:vmap     Visual
:nmap     Normal
:omap     Operator-pending
:map!     Insert and Command-line
:imap     Insert
:cmap     Command-line
```

Operator-pending 模式指这样一种情况： 你已经键入了一个作为命令的操作符,比如"d"或者"y"，接下来Vim希望继续键入一个移动命令或是一个文本对象.就是这种Vim希望继续接收命令而你又尚未键入的悬而未决

[1]译注： 对于MS-Windows用户来说，插入日期要这样： :read !date<NUL．因为date命令默认要求输入一个新日期．所以用<NUL重定向来打发它，缺点是命令返回值是1，你需要按一下<Enter>来继续；另一个平台无关的方法是使用vim的内置函数strftime，仿上例我们可以定义这样的映射： :imap <F2> <C-R>=strftime("%c")<CR>． 请参考|strftime()|

的状态，Vim术语里叫Operator-pending 模式． 比如对于命令"dw"，其中的"w"就是你在Operator-pending模式下键入的．

假设你想这样来映射<F7>: 使用d<F7>可以删除一个C语言的程序块(包括在花括号{}中的文本内容)．而y<F7>又可以yank这样一个程序块到无名寄存器中． 所以，你需要做的就是让<F7>来选择这样一个程序块:

ex command
```
:omap <F7> a{
```

这会让<F7>在Operator-pending模式下执行一个执行程序块的动作"a{"，就象你手工键入一样． 但对于键入{符号困难的键盘[1]来说这已经是不错的交易了．

列出映射

使用":map"(不带任何参数)可以让你查看当前就定义了哪些映射，或者，使用它的几种变体只列出指定模式下的映射． 输出看起来大致是这样的[2]:

List
```
   _g          :call MyGrep(1)<CR>
v  <F2>        :s/^/> /<CR>:noh<CR>``
n  <F2>        :.,$s/^/> /<CR>:noh<CR>``
   <xHome>     <Home>
   <xEnd>      <End>
```

列表的第一栏位指示出该映射工作于何种模式． "n"代表Normal模式，"i"代表Insert模式，诸如此类． 此栏空白的话是说这个映射是用":map"定义的，所以可以同时工作于Normal模式和Visual模式[3].[4]

查看当前映射列表有一个额外的收益: 它可以让你据此判断一些在<>中的字符是否能被Vim以特殊字符来对待(当然你的Vim要支持颜色高亮显示功能才行)． 比如说<Esc>以特殊颜色显示时，它表示ASCII为27的ESC键，如果以与普通文本相同的颜色显示，那它就只是5个普通字符的简单集合．

重映射

[5]

[1] 译注: 我没有见过键入{困难的键盘，只见过输入困难的人:-)

[2]译注: 本文中作为示例的输出往往依赖于你具体的环境，不要期望你自己的Vim执行起来总是跟这里的一样，本文中的其它部分不再对此作出说明

[3]译注: 这里说Visual模式同时指由命令"v"，"V" 和CTR-V进入的各种Visual子模式

[4]译注: 将"v"，"V"，CTRL-V进入的模式统称为Visual模式，而将这三种具体的Visual模式统称为Visual子模式，分别的称谓是"v""V"和"CTRL-V";作个比方，小明家有小明爸爸，小明妈妈，小明哥哥大明4个人，这4人统称为小明家,小明和他哥哥大明统称为小明家的孩子，这两个孩子又分别叫小明和大明

[5]译注: 英文是remapping，这里的re前辍并不表示重复做同一件事情，或覆盖上

Vim会检查一个映射的内容，看它是否包含了其它的映射． 比如上面对<F2>的映射也可以这样做：

ex command
```
:map <F2> G<F3>
:imap <F2> <Esc><F3>
:map <F3>  oDate: <Esc>:read !date<CR>kJ
```

Normal模式下的<F2>被映射为到最后一行，然后按下<F3>． Insert模式下的<F2>则退出Insert模式然后按下<F3>． 然后<F3>被映射为做这些具体的工作．

如果你几乎不怎么用Ex模式的话，你可以用"Q"命令来格式化文本(在Vim的老版本里它确实就是这个功能)：

ex command
```
:map Q gq
```

但是，偶尔你还是要进入Ex 模式． 我们可以把"gQ"映射为Q，这样还是可以通过它来进入Ex模式：

ex command
```
:map gQ Q
```

实际效果呢？你把"gQ"映射为"Q"． 这没什么错，但接下来你又把"Q"映射为了"gq"，所以"gQ"最终的结果还是"gq"，你还是不能进入Ex 模式．

要避免这种映射的内容又被映射的情况，用命令":noremap"命令：

ex command
```
:noremap gQ Q
```

现在Vim就知道说不要去检查"Q"是不是又被映射为了其它东西，其它模式下该命令的各种变体形式如下：

List
```
:noremap        Normal, Visual and Operator-pending
:vnoremap       Visual
:nnoremap       Normal
:onoremap       Operator-pending
:noremap!       Insert and Command-line
:inoremap       Insert
:cnoremap       Command-line
```

次做的结果，我个人更喜欢nest-mapping这样的表达，举个简单的例子:map <C-K> j<C-K> 这个映射之后你按下<C-K>光标会一遛跑到最后一行去，因为<C-K>执行完j之后又触发了一个<C-K>． 而用:noremap <C-K> j<C-K>则只会使光标下移一行，因为虽然它也在j之后触发了<C-K>，但这个<C-K>只会做它在VIM中的本职工作，而不会考虑它是否是一个映射． <C-K>中Vim的normal模式并不是一个命令，所以它实际上什么也不做．

当一个映射触发了它本身的执行，这个映射就会没完没了地映射下去． 这可用于无限次地重复一个操作．

比如说，你正在编辑一个文件列表，其中每个文件的每一行都含有一个版本号．以命令"vim *.txt"进入Vim编辑器． 现在你正在编辑的是第一个文件．定义下面的映射：

ex command
```
:map ,, :s/5.1/5.2/<CR>:wnext<CR>,,
```

现在按下",,"便会触发这个映射． 它把第一行中的"5.1"替换为"5.2"．然后在":wnext"保存文件并开始编辑下一个文件．但因为这个映射的最后是",,"． 这使得该映射象多米诺骨牌一样一路触发下去．

映射链会一直持续下去，直到碰到一个错误．错误可能是文件中找不到"5.1"． 如果是这样你可以稍事修改一下映射让它可以插入一个"5.1"并继续它的连锁反应．或者，错误的起因是":wnext"命令失败了，因为它已经达到文件列表的最后一个．

当一个映射在运行过程中发生了一个错误，那么映射的剩余动作就会被撤消掉．也可以按下CTRL-C来中断正在运行的映射(在MS-Windows上按CTRL-Break)．

删除一个映射

":unmap"命令可用于删除一个映射． 同样，对于各个不同的模式，这一命令有其相应的变体[1]：

List

:unmap	Normal, Visual and Operator-pending
:vunmap	Visual
:nunmap	Normal
:ounmap	Operator-pending
:unmap!	Insert and Command-line
:iunmap	Insert
:cunmap	Command-line

有一个小技巧可以定义一个映射同时在Normal模式和Operator-pending模式生效，但却对Visual模式无效． 首先定义它对三种模式都生效[2]，然后删除它在Visual模式下的映射．

ex command
```
:map <C-A> /---><CR>
:vunmap <C-A>
```

[1]译注： 规律:对于mapping簇的命令，一般来说都有其各个不同模式下的变体
[2]译注： Operator-pending可看作是Normal 模式的一个特例

注意"<C-A>"代表的是单个的按键CTRL-A.

|:mapclear|命令可用于删除所有的映射[1]. 可以想象，对不同的模式该命令又有类似的变体. 但是使用此类命令时可要小心，因为你无法撤消错误的操作.

特殊字符

":map"命令后面可以跟其它的命令. |字符是命令之间的分隔符. 同时这也暗示在一个映射的内容中不能出现|字符. 真要包括|字符的话，用<Bar>来代替它. 如:

ex command
```
:map <F8> :write <Bar> !checkin %<CR>
```

同样的问题也存在于":unamp"命令中，此外":unmap"命令还有一个特殊的问题：映射键的尾部空格. 下面是两个不同的命令:

ex command
```
:unmap a | unmap b
:unmap a| unmap b
```

第一个命令试图删除一个名为"a "的映射，这个映射以一个空格结束.

要在映射中使用空格的话，用<Space>来代替它:

ex command
```
:map <Space> W
```

这个命令使得空格键能让光标向前以word[2]为单位移动.

映射之后无法再加上注释，因为注释符号"也会被视为是映射内容的一部分. 你可以使用|符号来串起一个只带注释的空命令:

ex command
```
:map <Space> W|     " Use spacebar to move forward a word
```

映射与缩写

缩写很象Insert模式下的映射. 两个命令定义的形式也一样. 主要的区别在于两种功能发生作用的时机. 缩写功能在Vim识别出你已结束一个word时被触发. 而一个映射在你输入映射键时就会被触发.

[1]译注: 各个模式下的映射

[2]译注: 在Vim中术语中应该是大写的WORD，因为W不是根据\w来判断什么是一个word，而是依据\S，即非空白字符

两者的另一个区别是对于缩写，你在键入缩写字符的同时它就被插入在当前文本中． 一旦缩写被触发，缩写本身就会被它所对应的更长的内容所替换．而对于映射来说，你在键入的过程中，映射本身不会被插入到当前文本中，如果你打开了'showcmd'选项，键入的映射字符就会显示在Vim窗口的最后一行上．

下面是一个映射引起的具有歧义的情况． 假如你有下面两个映射：

ex command
```
:imap aa foo
:imap aaa bar
```

现在你按下"aa"，Vim就无从得知你是要使用第一个映射还是第二个．所以此时它还不能决定就此使用第一个映射，直到你键入第3个字符．如果是"a"，那它就会触发第二个映射，插入"bar"． 如果它是一个空格，那就使用第一个映射，插入的是"foo"，然后再加上你键入的空格[1]

更多...

<script>关键字可以让一个映射的作用范围局部于当前脚本． 请参考|:map-script|

<buffer>关键字可以让一个映射的作用范围局部于指定缓冲区，请参考|:map-buffer|

<unique>关键字可使定义一个即有的映射失败． 没有这个关键字时默认的情况是新定义的映射会覆盖旧的定义．请参考|:map-unique|．

要使一个映射什么事都不干，可以把它映射到<Nop>上去． 下面的命令使得<F7>什么都不做：

ex command
```
:map <F7> <Nop>| map! <F7> <Nop>
```

<Nop>后面必需没有任何空格．

40.2 自定义冒号命令

Vim编辑器允许你定义自己的冒号命令． 然后你就可以象执行Vim固有的冒号命令一样使用它．

[1]译注： 事实上不仅是空格，只要键入的第3个字符不是a，Vim都会判断使用第一个映射；另外，如果同时有缩写和映射，则Vim会优先使用映射，如现在同时":abbr aa tee"，则键入第一个a时Vim会等待一段时间看是否用户企图键入一个映射，超时后它会插入a，然后键入下一个a，此时Vim又会等待一段时间,如果超过了Vim对一个映射键的等待时间，则认为它不再可能是一个映射，接下来判断是不是一个缩写，对于缩写，Vim没有对键入每个字符的延迟进行监控，关于Vim对映射键的等待时间，请参考|mapping-typing|

要定义这样的命令，使用":command"命令[1]，如下例：

ex command
```
:command DeleteFirst 1delete
```

现在你使用":DeleteFirst"的话Vim就会实际执行":1delete"--删除第一行.

> 备注：自定义的冒号命令必需以一个大写字母开头. 但你不能使用":X"，":Next"和":Print"这些名字，也不能使用下划线！数字是允许的，但是不鼓励你使用.

下面的命令可列出所有使用自定义的冒号命令：

ex command
```
:command
```

自定义的冒号同样同Vim的内置冒号命令一样享有一些便利的特权. 只需键入足以区分不同命令的字符即可，此外也可以对其进行命令补齐.

参数的个数

用户自定义的冒号命令可以跟一系列的参数. 要指定它所能使用的参数，必需在定义时使用-nargs选项. 例如，上例中的:DeleteFirst命令没有参数，所以可以定义如下：

ex command
```
:command -nargs=0 DeleteFirst 1delete
```

不过，因为不跟参数是"command"命令默认的行为，所以并不必需用"-nargs=0". -nargs选项的其它可用形式如下：

List

-nargs=0	没有参数
-nargs=1	1个参数
-nargs=*	任意个数的参数
-nargs=?	0个或1个参数
-nargs=+	1个或多个参数

使用参数

在定义自己的冒号命令时，用关键字<args>来代表用户可能输入的参数：

[1]译注： ":command"可以看作是生产命令的超级命令

ex command
```
:command -nargs=+ Say :echo "<args>"
```

现在使用如下命令：

ex command
```
:Say Hello World
```

Vim就会显示"Hello World"． 不过你要是在参数中使用了双引号就会出问题，如下：

ex command
```
:Say he said "hello"
```

要把命令行上可能出现的特殊字符平顺地加入字符串中，可以使用<q-args>关键字：

ex command
```
:command -nargs=+ Say :echo <q-args>
```

现在再执行前面的":Say"命令就可以得到正确的结果了：

ex command
```
:echo "he said "hello""
```

<f-args>关键字包含的内容与<args>一样，不过它适用于把这些参数传递给一个函数调用，如下：

ex command
```
:command -nargs=* DoIt :call AFunction(<f-args>)
:DoIt a b c
```

上面两个命令等同于下面的命令行：

ex command
```
:call AFunction("a", "b", "c")
```

行号范围

一些命令以一个指定的范围使用它的参数． 要在Vim中定义这样的冒号命令，需要在定义时使用-range选项．该选项的可能取值如下：

List

-range	允许使用行号范围，默认是当前行
-range=%	允许使用行号范围，默认是所有行
-range={count}	允许使用行号范围，行号范围中的最后一行作为最后生效的单个数字，默认值是{count}

在定义命令时指定的-range选项的话，可以在被定义的命令实体中以<line1>和<line2> 这两个关键字来代表这个范围中的起始行和结束行. 例如，下面的定义的SaveIt命令，将把指定范围的行写入文件"save_file"中:

ex command
```
:command -range=% SaveIt :<line1>,<line2>write! save_file
```

其它选项

自定义命令时还有其它可用的选项和关键字，列表如下:

-count={number} 使命令可以接受一个命令计数作为参数，默认值是{number}. 在定义时可用<count>关键字来引用该数字

-bang 允许在定义的命令体中使用<bang>关键字来代替!

-register 允许把一个寄存器作为参数传递给该命令，命令体中对该寄存器的引用使用关键字<reg>或<register>

-complete={type} 定义该命令使用命令补齐的方式，请参考|:command-completion|了解该选项的可能取值

-bar 使该命令可以与其它命令共存于同一个命令行上，以|分隔，并且可以以一个"号进行注释

-buffer 使命令只对当前缓冲区生效.

最后要介绍的一个关键字是<lt>. 它代表字符<. 使用这个字符得以避免与关键字中使用的<符号相混淆.

重定义和删除命令

要重新定义一个命令只需要在command后面加一个!:

ex command
```
:command -nargs=+ Say :echo "<args>"
:command! -nargs=+ Say :echo <q-args>
```

要删除一个自定义冒号命令使用":delcommand". 它接受一个单一参数，作为要被删除的命令名. 例如:

ex command
```
:delcommand SaveIt
```

下面的命令删除所有的用户自定义命令:

ex command
```
:comclear
```

慎用! 这样的误操作可是不能恢复的!

关于自定义命令的详细内容请参考|user-commands|.

40.3 自动命令

一个自动命令是在某个事件发生时会自动执行的命令，可以引发一个自动命令被执行的事件包括读写文件或缓冲区内容被改变等等. 通过对自动命令的运用，你可以用Vim来编辑一个压缩过的文件，比如在|gzip|这个plugin中就用到了自动命令.

自动命令的功用非常强大，善用它们可以为你省去很多手工键入命令的麻烦. 同时也要小心，误用也可能给你带来极大的损失.

假如你希望每次保存文件时都会自动更新位于文件尾的时间戳. 首先你可以定义下面这样的一个函数:

ex command
```
:function DateInsert()
:   $delete
:   read !date
:endfunction
```

你希望在每次写文件时这个函数就会被调用，定义下面的命令:

ex command
```
:autocmd FileWritePre *  call DateInsert()
```

"FileWritePre"就是触发该命令自动被执行的事件: 就在写入文件文件之前. "*"是一个模式字串，用来匹配文件名，此例中要匹配的是所有文件.

打开这一命令后，每次你使用":write"命令，Vim都会触发以FileWritePre定义的自动命令，执行完之后再接着执行":write"操作.

:autocmd命令的一般形式如下:

ex command
```
:autocmd [group] {events} {file_pattern} [nested] {command}
```

其中的[group]名字是可选的. 它只是为了方便调用和管理这些命令(本文后面有更详细的解释). {events}参数是可以触发该命令的事件列表(每个事件以逗号分隔).

{file_pattern}是一个通常带有通配符的文件名． 象"*.txt" 使得被定义的自动命令对所有以".txt"的文件名生效． [nested]这个可选的标志允许对自动命令的递归触发.最后，{command}中的部分是实际上要执行的命令体.

事件

最常用的事件是BufReadPost． 它在每次Vim将要编辑一个文件时被触发． 通常用于设置选项的值．比如你知道"*.gsm"是GNU的汇编语言源文件．要为此类文件正确定义它的语法文件，可以用以下的自动命令:

ex command
```
:autocmd BufReadPost *.gsm  set filetype=asm
```

如果Vim能正常检测到文件的类型的话，它会设置'filetype'选项．这又会触发FileType 事件．这一事件可以让你为某一类的文件做些特别的设置．比如为所有的文本文件载入一个常用的缩写列表:

ex command
```
:autocmd Filetype text  source ~/.vim/abbrevs.vim
```

编辑一个新文件时，你也可以借助自动命令来让Vim为你自动生成一个框架.

ex command
```
:autocmd BufNewFile *.[ch]  0read ~/skeletons/skel.c
```

请参考|autocmd-events|了解可用事件的完整列表.

文件名模式

{file_pattern}参数由一系列以逗号分隔的模式组成． 比如"*.c,*.h"这样的模式匹配所有以".c"和".h"结尾的文件名.

通常的文件通配符都能用于这里的文件模式． 下面是一个常用文件模式通配符的列表:

List

*	匹配任意个数的任何字符
?	匹配一个任意的字符
[abc]	匹配字符a或b或c
.	匹配一个.号
a{b,c}	匹配ab和ac

如果文件名模式中含有斜杠(/)Vim就会比较目录名是否匹配． 没有斜杠的话只有文件名部分和命令中的文件名模式进行比较.例如"*.txt"可

以匹配"/home/biep/readme.txt".模式"/home/biep/*"也会匹配它，但"home/foo/*.txt"就不能匹配到该文件了.

包含斜杠时，Vim同时会检查文件的绝对路径名("/home/biep/readme.txt")和它的相对路径(例如"biep/readme.txt).

> 备注：对于一些以反斜杠作为目录之间的分隔符的系统，如MS-Windows来说，仍然可以在自动命令中使用正斜杠/. 这样写起文件名模式来更容易，也更容易在不同系统间移植你的脚本. 另外反斜杠不是还有它的特殊用途么.

删除自动命令

要删除一个自动命令，使用的形式与定义时相仿，只是不要再输入由{command}定义的命令体了，同时在autocmd后面加上一个!字符，如：

ex command
```
:autocmd! FileWritePre *
```

这将会删除所有使用文件名模式"*"的为"FileWritePre"事件定义的自动命令.

列出已定义的自动命令

[1]

下面的命令列出当前已定义的自动命令列表：

ex command
```
:autocmd
```

命令列表可能会很长，尤其是打开了文件类型检测功能时，要列表某一类的自动命令，可以在:autocmd的后面指定一个组名，事件名+文件名模式，或是文件名模式. 如下面的命令列出的是所有为BufNewFile定义的事件：

ex command
```
:autocmd BufNewFile
```

要列表所有为文件名模式"*.c"定义的自动命令使用：

ex command
```
:autocmd * *.c
```

[1]译注： 规律:主持一类功能的主命令名不加任何参数一般可以列出由它定义的子命令列表

在此使用的"*"代表所有的事件. 要列出所有cprograms组的自动命令, 使用:

ex command
```
:autocmd cprograms
```

命令组

{group}项用于在定义自动命令时为相关的命令分组. 也可用于在删除自动命令时以此为依据一次删除一批命令. 如下例.

要为某个命令组一次定义多个自动命令, 可以使用":augroup"命令, 下面我们来定义用于C程序的一些自动命令:

ex command
```
:augroup cprograms
:   autocmd BufReadPost *.c,*.h :set sw=4 sts=4
:   autocmd BufReadPost *.cpp   :set sw=3 sts=3
:augroup END
```

这等价于下面的形式:

ex command
```
:autocmd cprograms BufReadPost *.c,*.h :set sw=4 sts=4
:autocmd cprograms BufReadPost *.cpp   :set sw=3 sts=3
```

欲删除所有属"cporgrams"组的自动命令:

ex command
```
:autocmd! cprograms
```

嵌套自动命令[1]

一般情况下, 自动命令的命令体的执行不会再触发新的事件. 比如你在FileChangedShell 事件时执行的重新读文件内容的动作不会再去触发设置语法的事件, 要使它还能继续触发其它事件, 在autocmd中使用"nested"参数:

ex command
```
:autocmd FileChangedShell * nested  edit
```

强制执行自动命令

也可以强制执行一个自动命令, 就好象触发它的事件已经发生一样. 这对于在一个自动命令中想要触发另一个自动命令时很有用. 例如:

[1]译注: 这里的嵌套与map中的remap一样, 都是不准确的表达, 我喜欢称它为chained autocmd或recursive autocmd

ex command

```
:autocmd BufReadPost *.new execute "doautocmd BufReadPost ".expand("<afile>:r")
```

上例中定义了一个每次编辑新文件时都被触发的自动命令. 文件名必需以".new"结尾. ":execute"命令使用一个表达式来重新构建一个命令并执行它[1]. 如果新编辑的文件是"tryout.c.new"下面的命令就会被执行:

ex command

```
:doautocmd BufReadPost tryout.c
```

expand()函数以"<afile>"(意为触发autocommand[译注: a的由来]命令的文件名[译注: file的由来])作为参数, 并根据":r"标志取它的文件名部分[2].

":doautocmd"对当前缓冲区执行自动命令. ":doautoall"也一样, 不过它对每个缓冲区都分别执行自动命令.

使用Normal模式下的命令

被自动命令执行的命令都是冒号命令. 如果你想执行一个Normal模式下的命令, 可以使用":normal"命令. 如下例:

ex command

```
:autocmd BufReadPost *.log normal G
```

这将会使你在编辑以.log为扩展名的文件时光标自动定位到文件尾.

使用":normal"命令有一些小机关需要小心. 首先, 它假设后面的内容是一个完整的命令, 包括命令所需要的所有参数. 所以如果你用"i"命令来进入Insert模式, 你也必需以<Esc>离开Insert模式. 如果你用"/"命令来搜索一个字串, 该命令也必需以<CR>[3]结束.

":normal"命令把它后面的所有内容都看作是要在Normal模式下执行的命令. 这样一来就不能在该命令之后使用|来追加另一个命令了. 一定要这样做, 可以把":normal" 放在":execute"命令中, 这样做同时也使得一些不可打印字符更容易键入. 如下例:

ex command

```
:autocmd BufReadPost *.chg execute "normal ONew entry:<Esc>" |
        \ 1read !date
```

[1]译者: 相当于shell或perl中的eval函数

[2]译注: 此处说的文件名指去除扩展名之后的文件名, 对于象此例中的有多个点号的文件名, 最后一个点号前面的部分都被视为文件名, 虽然它也含有. 号

[3]译注: 回车键

此例也展示如何用反斜杠把一个长命令置放到多行上． 这一技巧可用于Vim脚本中(冒号命令中可不行)．

如果你要以自动命令做一些过于复杂的事情，比如要在文件中来回跳转并要求回到执行命令前的位置，你可能想要恢复整个文件在编辑现场的快照． 请查看|restore-position| 中的例子．

忽略一个事件

有时，你或许想避免触发一个自动命令． 'eventignore'选项中包含了一个Vim可以忽略的事件列表[1]．例如，下面的命令使进入和离开一个Vim窗口的事件被忽略．

```
:set eventignore=WinEnter,WinLeave
```

要忽略所有的事件，使用[2]：

ex command

```
:set eventignore=all
```

要恢复它的正常行为，可以通过清空'eventignore'中的事件列表来实现

ex command

```
:set eventignore=
```

下一章：|usr_41.txt| 写自己的VIM脚本
版　权： 请参考|manual-copyright| vim:tw=78:ts=8:ft=help:norl:

[1]译注： 规律:VIM中包含一个集合性质的数据项时都是逗号分隔，并且可以对该选项使用+=和-=来往集合中添加或删除元素

[2]译注： 对于集合性质的选项，一般都可使用特殊的关键字all代表所有的项，使用NONE代表一个空集合? 有时是，如hi ctermfg=none，但此处的事件集用空字串代表，参考下面．

usr_41.txt Vim 6.3版.最后修改: 2004年05月06日

VIM用户手册--- 作者: Bram Moolenaar

Vim脚本

Vim脚本语言用于Vim的初始化配置文件，语法高亮配置文件，以及诸多其它的地方.

下一章: |usr_42.txt| 添加菜单项
前一章: |usr_40.txt| 添加新的命令
目　录: |usr_toc.txt|

41.1 介绍 *vim-script-intro*

你与Vim脚本的第一次亲密接触将是Vim初始化配置文件，Vim在启动时读取该文件并执行其中的命令，你可以在此文件中设置你偏好的选项.也可使用任何的冒号命令(以一个冒号开始的命令；有时这些命令也被叫做Ex命令或命令行命令).

语法定制文件本质上也是Vim脚本，只是其中的选项集中为某种特定文件类型设置了特殊的值,为这种类型的文件在编辑、浏览方面提供额外的便利。也可以在一个独立的Vim脚本文件中定义一个复杂的宏，当然你可以随自己喜好扩展Vim脚本的其它应用。

让我们以一个例子开始Vim脚本的介绍:

code
```
:let i = 1
:while i < 5
:  echo "count is" i
:  let i = i + 1
:endwhile
```

备注: 冒号":"在此并不是必需的。只有你在Vim编辑环境中要使用此类命令时它们才是必需的，在Vim脚本文件中，冒号可要可不要。不过本文中将总是包含这些可选的冒号，以利于行文的清晰并且使它们区别于普通模式下的命令.

":let"命令为一个变量赋值，其一般形式如下:

code
```
:let {variable} = {expression}
```

本例中变量名为"i"，表达式是一个简单的数字值--1.

":while"命令开始一个循环. 其一般形式如下:

code
```
:while {condition}
:  {statements}
:endwhile
```

":while"语句和与它相匹配的":endwhile"之间的语句块在条件满足时被执行. 此处的条件是"i < 5". 该表达式在变量i的值小于5时为真.

":echo"命令打印显示其参数的值. 本例中要显示的参数是字符串"count is"和变量i的值. 变量i为1时，将显示:

Display
```
count is 1
```

接下来是一个":let i ="命令. 变量i将被赋的值是表达式"i + 1". 即将变量i的值加1然后把结果值重新赋予变量i本身.

本例的输出结果将是:

Display
```
count is 1
count is 2
count is 3
count is 4
```

备注: 如果你不幸写了一个停不下来的循环,可以按下CTRL-C键(在MS-Windows下按CTRL-Break)中断它.

数字值的三种表示法

数字值可以是十进制,十六进制或八进制. 一个十六进制数字以"0x"或"0X"打头. 如"0x1f" 值为31. 一个八进制数字以0打头. "017"值为15. 注意: 不要在希望使用十进制数字时以0打头,这样的数字序列将被解释为一个八进制数.

":echo"命令将总是以十进制输出. 如:

Display
```
:echo 0x7f 036
127 30
```

在一个数字值的前面放一个减号将使它成为负数. 前缀的减号同样可用于十进制,十六进制和八进制. 减号同样是减法表达式的操作符[1]. 将下式与上例比较:

Display
```
:echo 0x7f -036
97
```

表达式中的空格将被忽略. 而且,好的程序风格鼓励使用这样的空格去分隔不同的词法元素. 这样可增强表达式的易读性. 比如下例中在减号与数字之间放入一个空格可以避免将其看作一个负数:

Display
```
:echo 0x7f - 036
```

41.2 变量

一个变量名由ASCII字母,数字和下划线组成. 并且不能以数字打头. 合法的变量名形如[2]:

[1]译注: 此处放在数字前面的减号与作为表达式操作符的减号在程序语言的脚本具有不同的语意. 但为理解上的方便可以将它统一看作是减号

[2]译注: Vim脚本中变量沿用了经典的计算机语言中变量的词法定义,如C/C++/Java等.如果读者已熟知正则表达式,这一定义可表达式为"[a-zA-Z_][a-zA-Z0-9_]*"

Display
```
counter
_aap3
very_long_variable_name_with_underscores
FuncLength
LENGTH
```

非法的变量名形如"foo+bar", "6var". 这些变量都是全局变量[1]. 欲查看当前已经定义的所有变量,可以使用命令:

ex command
```
:let
```

可以在任意地方使用全局变量. 这意味着在一个脚本文件中引用的名为"count"的变量, 同样可以被另一个脚本文件所引用. 这样的应用至少会引起理解上的混乱, 实际情况往往更糟. 可以在变量名前前辍以"s:"使该变量局部有效于当前的脚本文件. 如一个脚本文件中含有如下代码:

code
```
:let s:count = 1
:while s:count < 5
:   source other.vim
:   let s:count = s:count + 1
:endwhile
```

由于变量"s:count"是局部于该脚本的,所以在另一个脚本如"other.vim"无论如何也不会触及到该变量的值.即使"other.vim" 也用到了一个名为"s:count"的变量. 那也是一个不同于前者的变量.局部于脚本"other.vim"自己. 关于局部于脚本的变量的更多内容,请参看|script-variable|

Vim脚本引入了几种不同的变量, 参见|internal-variables|. 其中最常用的是:

List

b:name	局部于一个缓冲区的变量
w:name	局部于一个窗口的变量
g:name	全局变量(同样适用于函数中)
v:name	Vim的预定义变量

删除变量

[1]译注: 通过本文的介绍, 读者将发现这些全局变量的概念与典型的程序语言中全局变量有所不同, 作为一个编辑器的伴生语言, 这里的全局变量还指在Vim的一个运行期内, 不同窗口, 不同缓冲区,不同的函数和不同的脚本文件中均引用同一变量

变量是占用内存空间的程序实体. Vim脚本中的变量可由":let"命令显示出来. 删除一个变量可用":unlet"命令, 如:

ex command
```
:unlet s:count
```

该命令删除了一个局部于脚本的变量"s:count"以释放占用的内存. 如果不知道是否存在某变量, 而且想在它不存在时避免看见错误信息, 可以使用!号:

code
```
:unlet! s:count
```

一个脚本执行完毕时, 其中的局部变量并不会自动释放. 下次执行该脚本时, 仍可继续使用该局部变量. 如:

code
```
:if !exists("s:call_count")
:  let s:call_count = 0
:endif
:let s:call_count = s:call_count + 1
:echo "called" s:call_count "times"
```

"exists()"函数检查一个变量是否已经被定义. 它的参数是以欲检查的变量的名字为内容的字符串. 而不是变量本身[1]! 如果使用以下命令:

code
```
:if !exists(s:call_count)
```

那么变量s:call_count的值将会作为exists()函数将检查的变量的名字. 这显然不是你要的结果.

警示符!对一个值取反. 如果当前值为true, 则以!取反后结果值为false. 反之, 值为false, 取反后为true. 该操作符可读为"not". 如此表达式"if !exists()"可看作"if not exists()". Vim眼中任何0之外的都是真值, 只有0是假值[2]

字符串变量和常数

到目前为止所讨论的变量值还只限于数值类型的. 字符串同样可用于变量值. 数字和字符串是Vim脚本所支持的仅有的两种数据类型. 变量的类型是动态确定的. 每次以":let"命令为变量被赋值时均会根据值的类型确定变量的类型. 为一个变量赋予一个字符串值, 可以使用一个字符串常数. 有两种情况, 第一种是字符串以双引号引起来:

[1]译注: 从程序语言的角度看, 变量名组成的字符串只是一个普通的字符串, 碰巧它的内容是程序符号表中某变量的名字. 而变量是指对应于内存中某处的值. 根据变量类型的不同和程序对该变量类型的解释. 该值可以有不同的解释.

[2]译: 与C语言的观点一样

ex command

```
:let name = "peter"
:echo name
peter
```

如果你要在字符串内容本身中包含双引号，要在"号前加一个\:

ex command

```
:let name = "\"peter\""
:echo name
"peter"
```

为避免使用反斜杠[1]，可以使用引号以避免作为字符串定界符的双引号与作为字符串内容的双引号的冲突[2]:

code

```
:let name = '"peter"'
:echo name
"peter"
```

单引号中的内容将完全按字面意思解释. 负面影响就是这样一来字符串的内容中不能含有单引号了. 反斜杠就是反斜杠，不再担当以特殊方式解释其后继字符的角色. 所以再不能使用特殊字符序列.

双引号中的字符串就不同了. 下面是一些在双引号中被另眼看待的字符序列:

List

```
t               <Tab>
\n              <NL>, 断行符
\r              <CR>, <Enter>
\e              <Esc>
\b              <BS>, backspace
\"              "
\\              \, backslash
\<Esc>          <Esc>
\<C-W>          CTRL-W
```

最后的两个例子代表了特殊情况. "\<name>"形式的表示法可以看作一个特殊健，该特殊健在Vim中以特殊的描述性名字表示.

参看|expr-quote| 了解字符串中特殊健表示法的完整内容.

[1]译注: 再好的打字员也讨厌敲\

[2]译注: 这种冲突会让语言解析器摸不着头脑，其它一些语言也支持以这种方式使双引号成为字符串内容的一部分，如bash，perl，php...

41.3 表达式

Vim以一种简单的方法处理丰富的表达式．参见|expression-syntax|了解表达式的定义．这里将会描述一些最常用的情形．

前面的数字值，字符串值本身就是表达式，所以凡是需要一个表达式的地方，都可以放入一个数字值或字符串．除此之外，表达式还可含有以下语素：

List
```
$NAME          环境变量名
&name          Vim中的选项名
@r             Vim中的寄存器名
```

例：

List
```
:echo "The value of 'tabstop' is" &ts
:echo "Your home directory is" $HOME
:if @a > 5
```

&name这种语法形式可用于保存一个选项值，为选项设置一个新的值，应用新的选项值进行一些操作后，再恢复该选项的初始值．如：

List
```
:let save_ic = &ic
:set noic
:/The Start/,$delete
:let &ic = save_ic
```

这将使模式"The Start"在选项'ignorecase'关闭的情况下被处理，操作完成后该选项又恢复了用户设置的值

数字运算

组合这些基本元素将会产生有趣的结果．我们以数字的数学运算开始：

List
```
a + b          加
a - b          减
a * b          乘
a / b          除
a % b          求模
```

运算的优先级使用普遍的数学法则，如：

code
```
:echo 10 + 5 * 2
20
```

使用括号对一个表达式分组以强制运算规则，如：

code
```
:echo (10 + 5) * 2
30
```

字符串可以以"."连接起来. 如：

code
```
:echo "foo" . "bar"
foobar
```

":echo"命令有多个参数时，它以空格分隔显示的多个参数的值. 本例中整个表达式作为一个参数，所以插入空格.

从C语言中借用的条件表达式形如：

code
```
a ? b : c
```

如果"a"求值为true则使用值"b"，否则使用值"c". 如：

code
```
:let i = 4
:echo i > 5 ? "i is big" : "i is small"
i is small
```

该表达式中的三个子表达式均会先被求值. 所以上式等价于：

code
```
(a) ? (b) : (c)
```

41.4 条件

只有当一个条件被满足时，":if"执行与其匹配的":endif"语句之间的语句，一般形式如下：

code
```
:if {condition}
   {statements}
:endif
```

只有表达式{condition}被求值为true(非0值)时语句{statement}才会被执行. 语句必需符合Vim脚本的语法规则，否则Vim将找不到标志语句块结束的关键字":endif".

同样可以有":else". 一般形式如下:

code
```
:if {condition}
   {statements}
:else
   {statements}
:endif
```

第二个语句{statement}只有在第一个语句不被执行时才被执行.

最后要介绍的是":elseif":

code
```
:if {condition}
   {statements}
:elseif {condition}
   {statements}
:endif
```

该关键字代替了两个独立的关键字":else"和"if"，并且，避免了使用额外的":endif".

用在Vim初始化脚本文件中一个有用的例子是检查'term'选项的值并依次做一些其它的设置:

code
```
:if &term == "xterm"
:  " Do stuff for xterm
:elseif &term == "vt100"
:  " Do stuff for a vt100 terminal
:else
:  " Do something for other terminals
:endif
```

逻辑操作

事实上，在上面的例子中已经用到了这些逻辑操作. 下面是最常用的逻辑操作符:

List

a == b	等于
a != b	不等于
a > b	大于
a >= b	大于或等于
a < b	小于
a <= b	小于或等于

或条件为真则逻辑操作的结果为1，否则为0． 如：

code
```
:if v:version >= 600
:   echo "congratulations"
:else
:   echo "you are using an old version, upgrade!"
:endif
```

这里"v:version"是Vim的预定义变量，它含有Vim的版本号．6.0版本的Vim版本号为600． 6.1版本的Vim版本号为601． 对于一个要在多个版本间共享的脚本，该变量非常有用．

逻辑操作符同时适用于数字值和字符串．当比较两个字符串时，使用字符串在数字表达上的差异进行判断． 即比较字符串中每个字符的ASCII码． 这对某些语系可能会造成歧义．拿一个字符串与数字进行比较时，字符串首先被转换为一个数字． 这其中有一些机巧，因为如果一个字符串看起来不象一个数字，它就会被转换为数字0． 如：

code
```
:if 0 == "one"
:   echo "yes"
:endif
```

这段脚本执行的结果将是显示"yes"，因为"one"看起来不象一个数字，所以它被转换为数字0，实际进行的是两个数字0之间的比较．

对字符串来说还有两个额外的操作符：

List

a =~ b	a包含b
a !~ b	a中不包含b

左边的a作为一个字符串，右边的b被视为一个查找模式，就象在a中搜索b，如：

code
```
:if str =~ " "
:  echo "str contains a space"
:endif
:if str !~ '\.$'
:  echo "str does not end in a full stop"
:endif
```

注意上例中如何使用单引号来指定一个搜索模式. 这非常有用! 因为在一个以双引号括起来的字符串中,要表达一个真正的反斜杠,就必需写连续的两个反斜杠,在写一个搜索模式字符串时,这可能要写很多个反斜杠.

比较两个字符串时选项'ignorecase'的设置影响比较操作. 如果想摒除该选项的影响,可以使用后缀的"#"进行大小写敏感的比较,后缀以"?"进行忽略大小写的比较. 这样"==?"操作符将忽略大小写进行字符串的比较. 而"!~#"检查两个字符串是否不相同时,区别对待大小写不同的字母. 欲知该操作的详情,参见|expr-==|.

更多的循环相关操作符

前面已经提到":while"命令. 另有两个语句可用于":while"和":endwhile"之间:

code
```
:continue               跳转到loop循环的开始,循环继续
                        loop continues.
:break                  向前跳转到":endwhile";循环中断
```

如:

code
```
:while counter < 40
:  call do_something()
:  if skip_flag
:    continue
:  endif
:  if finished_flag
:    break
:  endif
:  sleep 50m
:endwhile
```

":sleep"命令会让Vim小睡片刻. "50m"指50毫秒[1],如果是:sleep 4,则让Vim睡眠4秒.

[1] 译注: 1秒等于1000毫秒

41.5 执行一个表达式

目前为止提到的命令都是被Vim直接执行. ":execute"命令允许以一个表达式的值作为要执行的命令. 该命令提供了动态构建一个命令的强大功能.

使用该命令的一个例子是跳转到一个指定的标签处, 该标签的名字为某变量的值:

ex command
```
:execute "tag " . tag_name
```

"."用于连接字符串"tag "和变量"tag_name" 的值.假设变量"tag_name"的值为"get_cmd". 则最终被执行的命令是:

ex command
```
:tag get_cmd
```

":execute"命令只能用于执行"冒号命令". ":normal" 使用可用于执行Normal模式下的命令. 不过,它的参数不能是一个表达式而必需是在Normal下被逐字解析执行的命令序列. 如:

ex command
```
:normal gg=G
```

该命令跳转到当前缓冲区的首行并用"="操作格式化所有的行.

要使":normal"命令借用表达式的灵活性. 可以组合使用":execute". 如:

ex command
```
:execute "normal " . normal_commands
```

变量"normal_commands"的值必需是合法的Normal模式命令.

注意保证":normal"的完整命令名都被键入.否则Vim将一直解析到参数列表的末尾并终止该命令. 如下面命令进入插入模式.命令结束前必需离开插入模式:

ex command
```
:execute "normal Inew text <Esc>"
```

该命令在当前行的前面插入字符"new text". 注意此处如何指定特殊健ESC "\<Esc>".这避免了在脚本中嵌入一个真正的<Esc>字符[1].

[1]译注:嵌入一个真正的<Esc> 意味着在文本文件组成的脚本中插入一个ASCII值为27的字符, Vim中将以^[的方式将其显示为可见字符. 在以COPY & PASTE方式处理此类脚本时, ^[将被剪贴板处理为两个字符^和[, 引起脚本错误

41.6 使用函数

Vim定义了很多函数以提供丰富的功能. 本节中将会有一些使用这些函数的示例. 可以在|functions|发现函数的完整列表.

一个函数以":call"命令调用. 被传递的参数放在两个括号中, 依序以逗号分隔, 如:

ex command
```
:call search("Date: ", "W")
```

此例将以参数"Date: "和"W"调用search()函数. search()函数以第一个参数作为一个搜索模式并以第二个参数作为旗标值修饰搜索操作的某些细则. 旗标值"W"意味着搜索操作在到达文件尾将不会回绕到文件头继续搜索.

一个函数调用也可以出现在一个表达式中, 如:

ex command
```
:let line = getline(".")
:let repl = substitute(line, '\a', "*", "g")
:call setline(".", repl)
```

getline()函数从当前文件中得到一行. 它的参数以某种方式确定要操作的行. 本例中使用".", 这一特殊指示符代表光标所在的当前行.

substitute()函数执行的操作类似于":substitute"命令. 第一个参数是要进行操作的字符串. 第二个参数是搜索的模式,第三个参数指定找到的搜索模式将被替换为的目标字符串.最后一个参数是操作的旗标修饰符.

setline()函数将第一个参数指定的某行的内容置为由第二个参数指定的值.本例中当前光标下的行的内容被替换为substitute()函数的结果值.所以上述操作等价于:

ex command
```
:substitute/a/*/g
```

当你要在调用substitute()前后做更多的操作时使用函数会更有趣.

函数

function-list

很多, 很多函数. 在此我们将一一点提它们. 这些函数将根据其功能分组. 不过你也可以在|functions|找到一个字母顺序的函数列表. 另外, 当光标停在函数名上时使用CTRL-]可以跳转到该函数的详细帮助上.

字符串操作:

List

char2nr()	得到一个字符的ASCII码值
nr2char()	得到一个ASCII值对应的字符
escape()	返回一个字符串以\转义符表达式的形式
strtrans()	将一个字符串转换为可显示形式[[译注：如将ASCII为9的TAB键字符显示为^I]
tolower()	将一个字符串转换为小写
toupper()	将一个字符串转换为大写
match()	返回一个搜索在一个字符串中出现的位置
matchend()	同match()，但从后往前搜索
matchstr()	同match，但返回匹配到的目标串
stridx()	一个子串初次出现在母串中的位置
strridx()	一个子串最后一次出现在母串中的位置
strlen()	求字符串长度.
substitute()	字符串替换
submatch()	得到一次":substitute"操作中的匹配到的一个子模式[[译者：指用\(\)括起
strpart()	得到一个字符串的子串
expand()	扩展特殊的关键字
type()	变量的类型，返回0为数字，1为字符串[[译者：注意其参数要求是变量名没错
iconv()	转变文字编码

当前缓冲区操作:

List

byte2line()	得到该文件中第N个字节位于第几行
line2byte()	同上面的相反，求得指定行的首字符位于文件中的字节偏移
col()	得到光标或一个mark所在的列号
virtcol()	得到光标或一个mark所在屏幕上的列位置
line()	得到光标或一个mark所在的行号
wincol()	当前光标在当前窗口坐标中的列号
	[[译者： 如果你打开了'number'选项，则得到的值会比
winline()	当前光标在当前窗口坐标中的行号
cursor()	将光标置于指定的行列
getline()	从当前缓冲区中得到指定行的内容
setline()	将缓冲区某行的内容替换为指定内容
append()	将参数中指定的字串附加在指定行的下面
indent()	得到指定行的缩进量
cindent()	indent according to C indenting
lispindent()	indent according to Lisp indenting
nextnonblank()	返回指定行及指定行之后的第一个非空行
prevnonblank()	返回指定行及指定行之前的第一个非空行
search()	查找下一个正则表达式
searchpair()	find the other end of a start/skip/end

系统函数和文件操作:

List

browse()	打开一个文件对话框请求指定一个文件
glob()	扩展文件名通配符
globpath()	同上，但可以指定一系列的目录
resolve()	返回MS-Windows上一个链接所指向的目标
fnamemodify()	返回文件名的各种形式 [[译者：如全路径名，文件名,扩展名等]]
executable()	检查一个可执行程序是否存在
filereadable()	检查文件的可读性
filewritable()	检查文件是否可写
isdirectory()	检查一个目录是否存在
getcwd()	返回当前工作目录
getfsize()	得到指定文件的大小
getftime()	得到指定文件的最后修改时间
localtime()	获取当前时间
strftime()	将时间转换为字符串形式
tempname()	得到一个临时文件名
delete()	删除文件
rename()	文件更名
system()	执行一个SHELL命令
hostname()	当前系统的主机名

缓冲区，窗口和参数列表：

List

argc()	返回参数列表中的参数个数
argidx()	参数列表中的当前索引号
argv()	返回参数列表中指定索引号的参数
bufexists()	检查一个缓冲区是否存在
buflisted()	检查一个缓冲区是否存在并且位于缓冲区列表中
bufloaded()	检查一个缓冲区是否存在并被载入
bufname()	返回指定缓冲区号所对应的缓冲区名字
bufnr()	返回指定缓冲区名字所对应的缓冲区号
winnr()	返回当前窗口的窗口号
bufwinnr()	得到指定缓冲区所在的窗口号，没有对应窗口时返回 -1
winbufnr()	返回指定窗口中所编辑的缓冲区的号码
getbufvar()	得到指定缓冲中的一个变量
setbufvar()	为指定缓冲区设定变量值
getwinvar()	得到指定窗口的变量
setwinvar()	为指定窗口设置变量

折行：

List

foldclosed()	检查指定行是否位于一个处于关闭状态的折行内，若是返回该折行的首行行号，否非，返回-1
foldclosedend()	同上，返回的是折行的尾行行号
foldlevel()	返回指定行的折行层级，没有定义折行时返回0
foldtext()	生成一个关闭的折行所显示的文本[[

语法高亮：

List

hlexists()	检查一个语法高亮项是否存在
hlID()	返回一个语法高亮项的ID
synID()	返回指定位置的语法ID
synIDattr()	返回某语法ID的指定属性
synIDtrans()	得到某语法项最终应用了其颜色定义的那种语法的ID

历史列表：

List

histadd()	向历史列表中加入条目
histdel()	从历史列表中删除条目
histget()	从历史列表中得到一个条目
histnr()	得到历史列表中记录个数

交互操作：

List

confirm()	让用户确认一组选择
getchar()	请用户输入一个字符
getcharmod()	get modifiers for the last typed character
input()	从用户那要一行内容
inputsecret()	同上，但不回显输入的内容
inputdialog()	让用户在一个对话框里输入
inputresave	save and clear typeahead
inputrestore()	restore typeahead

Vim服务器：

List

serverlist()	返回服务器名字列表
remote_send()	向指定服务器发送命令字符
remote_expr()	在服务器求值一个表达式
server2client()	向客户端发送一个回应
remote_peek()	检查是否收到来自服务器的回应
remote_read()	从服务器读取回应信息
foreground()	将Vim窗口移到前台
remote_foreground()	将Vim服务器窗口移至前台

其它:

List

mode()	返回当前的编辑模式
visualmode()	返回最近一次使用的Visual子模式
hasmapto()	检查是否有一个键被定义为指定的内容
mapcheck()	检查一个映射是否存在
maparg()	返回一个映射的右部，即它所被映射的内容
exists()	检查一个变量或函数等是否被定义
has()	检查当前的Vim是否支持某个特性
cscope_connection()	检查是否存在某个cscope连接
did_filetype()	检查是否一个设置文件类型的自动命令已被执行过
eventhandler()	检查当前脚本是否因事件触发而被调用
getwinposx()	返回窗口的X坐标，单位为象素
getwinposy()	返回窗口的Y坐标，单位为象素
winheight()	返回指定窗口的高度，单位为字符
winwidth()	返回指定窗口的宽度，单位为字符
libcall()	调用外部共享库中的一个函数
libcallnr()	同上，但用于返回int的函数

41.7 函数定义

Vim允许用户自定义函数. 基本的函数定义如下:

code
```
:function {name}({var1}, {var2}, ...)
:   {body}
:endfunction
```

备注: 函数名必需以一个大写字母开始.

我们来定义一个简单函数，该函数返回两个数中较小的数，该函数的定义以以下命令开始:

code
```
:function Min(num1, num2)
```

这告诉Vim定义一个名为"Min"的函数，该函数接受两个参数"num1"和"num2".

函数体中第一件事就是检查两个参数哪个更小:

code
```
:   if a:num1 < a:num2
```

特殊的前辍"a:"告诉Vim该变量是一个函数参数[1]，下面我们把较小的值赋予变量"smaller"：

code
```
:   if a:num1 < a:num2
:     let smaller = a:num1
:   else
:     let smaller = a:num2
:   endif
```

变量"smaller"是一个局部变量． 在一个函数中使用的变量将局部于该函数，除非该变量前辍以"g:"，或"a:"，或"s:"．

备注: 要在一个函数中引用一个全局变量必需前辍以"g:"． 这样在一个函数中引用"g:count"指对全局变量"count"的使用．而对"count"存取实际操作的是另一个局部于该函数的变量．

现在可以使用";return"语句返回两个数中较小的值．最后，结束这个函数定义：

code
```
:   return smaller
:endfunction
```

完整的函数定义如下：

code
```
:function Min(num1, num2)
:   if a:num1 < a:num2
:     let smaller = a:num1
:   else
:     let smaller = a:num2
:   endif
:   return smaller
:endfunction
```

一个用户自定义的函数的调用与对Vim内置函数的调用毫无二致，唯一的不同是函数名的命令规范，用户自定义函数必需以大写字母开始，上例中的Min函数可以这样应用：

code
```
:echo Min(5, 8)
```

[1]译注： a代表argument

直到现在函数才会被执行，函数体被Vim解释，如果其中有误，比如使用了未定义的变量或调用了未定义的函数，Vim将会发出一个错误信息．而定义函数时这些错误还不能被检测到．

当一个函数执行到":endfunction"语句时或者":return"以不带参数的形式返回． 则函数返回值为0．

要重新定义一个已经定义过的函数名，在函数定义命令":function"后附一!号[1]

code
```
:function!  Min(num1, num2, num3)
```

在函数中使用行号范围

":call"命令可以指定一个函数操作将施于其上的行的范围．此类操作有两种可能的执行期语意． 当一个函数定义时指定了"range"关键字时，函数将自己处理给定的行的范围．函数被调用时将被调用者以变量"a:firstline"和"a:lastline"传入行范围的上下限．如:

code
```
:function Count_words() range
:   let n = a:firstline
:   let count = 0
:   while n <= a:lastline
:     let count = count + Wordcount(getline(n))
:   endwhile
:   echo "found " . count . " words"
:endfunction
```

该函数可以这样调用:

ex command
```
:10,30call Count_words()
```

函数将被调用一次，显示出指定范围的文本行中的单词数． 使函数作用于一定范围的文本行的另一种方法是定义函数时不指定"range"关键字． 这样该函数将于指定范围中每一个文本行被调用一次． 如:

code
```
:function  Number()
:   echo "line " . line(".") . " contains: " . getline(".")
:endfunction
```

如果这样调用该函数:

[1]译注： Vim中相同函数名的函数将被视为同一个函数，大小写敏感，如C语言

ex command
```
:10,15call Number()
```

函数Number()将被执行6次.

未知个数的函数参数

Vim允许函数接受个数可变的参数，比如下面的命令中，定义了一个至少有一个参数的函数，它最多可接受20个参数:

code
```
:function Show(start, ...)
```

变量"a:1"代表第一个可选参数，"a:2"指第二个，依此类推. 变量"a:0"包含了实际传递的参数的个数. 如:

code
```
:function Show(start, ...)
:  echohl Title
:  echo "Show is " . a:start
:  echohl None
:  let index = 1
:  while index <= a:0
:    echo "  Arg " . index . " is " . a:{index}
:    let index = index + 1
:  endwhile
:  echo ""
:endfunction
```

该例中使用":echohl"命令指定接下来的":echo"命令所使用的语法高亮规则. ":echohl None"禁止语法高亮. ":echon"命令象":echo"命令一样输出参数的内容，区别只是它并不象后者一样在输出结束时换行.

显示已定义的函数

":function"命令将显示用户自定义函数的列表:

code
```
functions:
        :function
        function Show(start, ...)
        function GetVimIndent()
        function SetSyn(name)
```

要想知道一个函数到底做了什么，可以以其函数名作为":function"命令的参数:

Display
```
:function SetSyn
1      if &syntax == ''
2        let &syntax = a:name
3      endif
endfunction
```

函数调试

在你遇到错误或进行调试时行号信息是十分可贵的，参考|debug-scripts|进一步了解调试模式.也可以通过将'verbose'选项设为12或一个更大的数来查看所有的函数调用． 将它设为15或更多大可以看到每一个被执行的命令行.

删除函数

要删除函数Show()，可用命令:

ex command
```
:delfunction Show
```

如果此时该函数还不存在，Vim将返回一个错误信息.

41.8 异常

先看一段代码:

code
```
:try
:   read ~/templates/pascal.tmpl
:catch /E484:/
:   echo "Sorry, the Pascal template file cannot be found."
:endtry
```

上例中如果文件不存在":read"命令就会失败． 这时代码会更友好地提示用户而不是给出一个错误信息.

由":try"和":endtry"之间的命令所产生的错误都会被转为异常． Vim中的异常是一个字符串．如果异常是由错误所引起的那么这个字符串的内容就是错误信息．每条错误信息都对应一个数字． 此例中我们捕捉到的错误含有"E484:"． 该错误号是固定不变的(相对而言，错误信息本身很可能会有不同，比如文档被译为其它语言).

如果":read"引发了另外的错误，那么"E484:"这个模式匹配不到它了．这时异常就不会被捕捉到，结果是仍按普通的错误进行处理．

或许你会尝试下面的代码：

```
code
:try
:   read ~/templates/pascal.tmpl
:catch
:   echo "Sorry, the Pascal template file cannot be found."
:endtry
```

这会捕捉到所有的错误．但如此一来你也就看不到那些有用的错误信息了，比如象"E21: Cannot make changes, 'modifiable' is off"．

另一个有用的异常机制是":finally"命令：

```
code
:let tmp = tempname()
:try
:   exe ".,$write " . tmp
:   exe "!filter " . tmp
:   .,$delete
:   exe "$read " . tmp
:finally
:   call delete(tmp)
:endtry
```

这段代码会通过一个"filter"命令来过滤自当前光标所在行到文件尾的内容，该命令跟有一个文件名作为参数．无论过滤程序是否能正常执行，只要":try"和":finally"之间的代码产生了错误，或者用户通过CTRL-C 撤消了操作，":call delete(tmp)"这个命令都会被执行．这就保证了不会留下弃之不用的临时文件．

关于异常处理的更多信息请参考|exception-handling|．

41.9 注意事项

此处列出一些与Vim脚本相关的注意事项．这些相关事项也会在该文档的其它地方提及．但这里将对此作一总结：

行结束符与具体的系统有关．对Unix而言是一个单个的<NL>字符[1]．

[1]译注： NL代表New Line.其ASCII值为10，在计算机语言中通常表达为"\n"

对MS-DOS，Windows或OS/2此类系统，用的是<CR><LF>[1].当使用一些以<CR>为结束的mapping时这一约定就显得十分重要了． 参见|:source_crnl|

空白符

Vim脚本中的空白行是允许的，在执行脚本时这些空白行将被忽略．

一行文本实际内容之前的任意个空白字符(空格键和跳格键)总是被忽略掉．而在命令与其参数之间的多个空白字符将总被看作单个的空格字符，作为命令与参数或参数之间的分隔符，一个命令行中最后一个可见字符之后的空白字符可能被忽略也可能不被忽略，视情况而定，继续往下看．

如下的":set"命令中用到了"="符号：

ex command

```
:set cpoptions    =aABceFst
```

出现在"="号之前的空白字符会被忽略．而"="之后可不能再出现空格!

要把空格作为要设置的值的一部分，必需使用"\"：

ex command

```
:set tags=my nice file
```

如果上面的命令写作：

ex command

```
:set tags=my nice file
```

Vim将会报告一个错误，因为该命令将被解释为[2]：

ex command

```
:set tags=my
:set nice
:set file
```

注释

[1]译注： CR代表Carriage Return回车，LF代表Line Feed，送纸，这一术语源自计算机的史前时代打字机的操作．当一个打印行已到打印纸右边缘时,打字机用于打字的笨重铁头先是回到当前打印行的最左边． 进行这一操作时打印纸不动．然后打字机将打印纸向前送出一行． 开始下一行的打印.回车换行两个独立的操作源于物理世界中机械操作的局限.对于显示在显示器上的文本行而言． 回车换行经常被理解为一个操作．

[2]译注： 假如nice与file碰巧是Vim中的选项名，将不会有错误报告，但这将不反映上面命令的原意

双引号"标志着注释的开始. "之后的任何字符(包括"自身)直到行尾都被视为注释，不过凡事都有例外，下面的例子中有些命令根本不考虑注释，它将命令名至行尾的所有内容都看作是命令的一部分. 此类情况除外，一个注释可以出现在一行脚本中的任何位置.

有些命令根本不鸟注释，如:

List
```
:abbrev dev development          " shorthand
:map <F3> o#include              " insert include
:execute cmd                     " do it
:!ls *.c                         " list C files
```

被定义的缩略词'dev'的内容将是'development " shorthand'. 而<F3>将被映射为自o#include至行尾的全部内容，包括'" insert include'. "execute"命令会引发一个错误. "!"命令会把此后的所有内容一鼓脑送到shell. 因为一个不匹配的双引号，此处也将引发一个错误[1].

":map", ":abbreviate", ":execute"和"!"命令之后不允许注释(此外还有少数几个命令也是如此). 不过对":map", ":abbreviate"和":execute"命令有一个变通办法:

List
```
:abbrev dev development|" shorthand
:map <F3> o#include|" insert include
:execute cmd                    |" do it
```

"|"字符用于分隔多个命令. 此处被分隔的第二个命令的全部内容就是一个注释.

注意在被定义的缩写和映射键中使用"|"时字符"|"不要有任何空白，因为对此类命令而言"|"之前的任意东西都被视为命令的一部分. 由于这些命令的这一特性. 下面命令中的不可见空白字符实际上也是命令的一部分:

ex command
```
:map <F4> o#include
```

为避免此类问题，可以在编辑你的Vim初始化文件时打开'list'选项

缺陷

问题还不止于此:

[1]译注:这一错误是由shell解释该命令行时发生的，而不是Vim处理该命令行时发生的，Vim对该命令行的处理将只是简单地把它送给shell

ex command
```
:map ,ab o#include
:unmap ,ab
```

此处的unmap命令将不能正常工作，因为它要unmap的字符序列是",ab ". 而这个字符序列并未被map命令映射. Vim将报告一个错误，当然很难弄明白原因，因为":unmap ,ab "结尾的空白字符是不可见的.

在'unmap'命令中使用一个注释时也会引发同样的问题:

ex command
```
:unmap ,ab      " comment
```

此处的注释部分被忽略. 而且Vim实际要unmap的东西是",ab ", 当然这一映射不存在. 这个命令可以这样表达:

ex command
```
:unmap ,ab|" comment
```

恢复视图

有时候你在编辑过程经常希望到文本的另一位置作一些修改后，回到刚刚离开的地方. 如果能精确地重现离开某位置时的现场那真是太棒了.

下面的例子将把当前行的内容复制到文件的最开始处，然后恢复此操作之前的屏幕现场:

ex command
```
map ,p ma"aYHmbgg"aP`bzt`a
```

映射的,p按键序列将执行下列操作:

List
```
ma"aYHmbgg"aP`bzt`a
ma                       在当前位置设置标签a
  "aY                    将当前行的内容复制到寄存器a中
     Hmb                 当光标定位到当前窗口的第一行并在此设置标签b
        gg               定位到文件的第一行
          "aP            把刚才复制的文本行放到第一行的前面
             `b          回到刚才离开时显示窗口的第一行
               zt        把该行置为当前窗口的首行，就象离开时一样.
                 `a      将光标定位到进行整个操作之前的位置
```

包

为避免你在自己定义函数名时与其它的函数名发生冲突，可以使用下面的方案:

- 在每个函数名的前面以一个唯一的字符串作为函数名的前辍.
  我自己经常使用一个代表某种意义的缩写词. 如"OW_"用于所有的与窗口相关的函数.
- 把你自己定义的函数集中放在一个脚本文件中.
  设置一个全局变量以标志你的函数定义是否已经被Vim载入[1]. 下次再执行该文件时, 可以据此先unload这些函数定义.

如:

code
```
" This is the XXX package

if exists("XXX_loaded")
  delfun XXX_one
  delfun XXX_two
endif

function XXX_one(a)
        ... body of function ...
endfun

function XXX_two(b)
        ... body of function ...
endfun

let XXX_loaded = 1
```

41.10 定制自己的plugin *write-plugin*

你可以写这样一种脚本: 其它的Vim用户把你的脚本文件放入它们的plugin目录, 马上就能使用其中的功能, 这种脚本叫plugin. 参见|add-plugin|

共有两种类型的plugin:

全局plugin: 对所有类型的文件生效.

filetype相关的plugins: 只会应用到特定类型的文件上.

本节中将讨论第一种类型的plugin. 写一个filetype相关的plugin涉及很多因素, 下节将讨论这一主题|write-filetype-plugin|.

[1]译注: 此处载入指Vim读取磁盘文件中的函数定义, 并将定义的函数加入已定义的函数表中

名字

首先要为自己定制的plugin起一个名字. 名字应该能让人望文生义，同时避免选取这样的名字:其它plugin已经使用了类似的名字，但实际上执行的操作却与你将要定制的操作大相径庭. 另外，为避免在老式Windows上引起问题，限制plugin的名字在8个字符以内[1]. 执行纠错功能的脚本可以叫"typecorr.vim". 这里我们就以此为例.

写一个人人能用的plugin有一些规则必需遵循. 我将一步步解释这些规则. 一个plugin的完整例子列在本文的最后.

正文

我们以plugin的正文开始，下面是执行实际操作的命令:

ex command

```
14      iabbrev teh the
15      iabbrev otehr other
16      iabbrev wnat want
17      iabbrev synchronisation
18              \ synchronization
19      let s:count = 4
```

当然，现实世界中的脚本要比这里列出的长多了.

这里出现的行号为了解说的便利，写你自己的plugin可别把它放在你的脚本里!

头部

一旦你写了一个脚本，很可能接下来你会对该脚本做一些修改，然后这个脚本就有了几个不同的版本. 所以当你公开发布你的plugin脚本时，他人可能会想知道是谁写了这么好的东东，或者，他也可以把赞扬或批评发给你. 所以，最好在你的plugin脚本的开头放上类似于这样的注释:

Display

```
1       " Vim global plugin for correcting typing mistakes
2       " Last Change:  2000 Oct 15
3       " Maintainer:   Bram Moolenaar <Bram@vim.org>
```

关于版权和许可证: 由于plugins非常有用同时又难以限制它的传播，所以请你把自己的plugins置为公共域软件或者使用Vim版权|license|. 象上例中的一个简短声明就足够了:

[1]译注: 包括8个字符

Display
```
4        " License:        This file is placed in the public domain.
```

续行问题，避免副作用

use-cpo-save

上面例子中的第18行代码中，一行内容被延续到下一行继续，这种机制在Vim字典里叫|line-continuation|. 打开了'compatible'设置的用户可能会因此遇上麻烦，Vim将对此报告一个错误. 但是不能简单地关闭'compatible' 选项，因为这会带来很多的副作用.解决这个问题可以把'cpoptions'选项临时设为它的默认值，之后再恢复它. 这样可以利用续行机制的好处同时让脚本能在不同的用户设置环境中都正常工作. 具体做法如下：

code
```
11      let s:save_cpo = &cpo
12      set cpo&vim
..
42      let &cpo = s:save_cpo
```

首先我们把'cpoptions'选项的值保存在变量s:save_cpo中，plugin结束时再恢复它.

注意这里对局部于脚本的变量|s:var|的使用. 使用全局变量的话，变量有可能已经在别的地方被用过了. 所以做这种针对脚本的操作时尽量使用这种局部于脚本的变量.

避免载入

可能用户并不问题想载入某个plugin. 或者系统管理员把某个plugin放在了一个公共目录，但是用户有他自己版本的plugin. 所以用户应该有机会避免载入某个plugin. 下面的脚本可以处理这种情况：

code
```
6       if exists("loaded_typecorr")
7         finish
8       endif
9       let loaded_typecorr = 1
```

这种做法同时避免了同一脚本被载入两次，一个脚本被载入一次以上可能会引起错误，比如重新定义函数引起的错误，多次定义autocommand引起的错误.

映射

现在我们来把plugin弄得更有趣一点：在plugin中加入拼写校正．当然可以直接写一个mapping．但mapping的名字可能已被使用了．为避免重定义一个在别处已经定义过的mapping，可以在map命令中使用<Leader>

22 map <unique> <Leader>a <Plug>TypecorrAdd

"<Plug>TypecorrAdd"执行真正的动作．

用户可以将变量"mapleader"设置为一个新的值，这样被map的键序列就必需以该值开头才会生效．如下：

```
code
let mapleader = "_"
```

该命令将定义一个名为"_a"的map．如果没有设置变量"mapleader"，则使用其默认值--反斜杠．即"\a"将被map．

注意这里使用了<unique>，这使得重定义一个键序列时Vim报告错误．|:map-unique|

但是如果用户想定义自己的健值序列呢？看下面的命令：

```
code
if !hasmapto('<Plug>TypecorrAdd')
  map <unique> <Leader>a  <Plug>TypecorrAdd
endif
```

这段脚本检查是否已经有一个键被映射为了"<Plug>TypecorrAdd"，只有在不存在这样的map时，才重新定义"<Leader>a"．用户可以把这段代码安全地放入自己的初始化脚本中：

```
Display
map ,c  <Plug>TypecorrAdd
```

这样被映射的键将是",c"而不是"_a"或"\a"．

分块

脚本变得越来越长时，用户往往希望把它们分隔为小的代码块．可以用函数和map来做到这一点．但是引入函数和map又可能和其它脚本中的同名元素发生冲突．比如，你定义了一个函数Add()，而另一个脚本也试图定义一个同名的函数．为避免这种情况,可以在函数名前面加上"s:"以使该函数局部有效于当前脚本[1]．

下面新增的函数执行一个新的拼写校正：

[1]译注： s代表script

code

```
function s:Add(from, correct)
  let to = input("type the correction for " . a:from . ": ")
  exe ":iabbrev " . a:from . " " . to
..
endfunction
```

现在可以在脚本里调用函数s:Add(). 如果另一个脚本也定义了一个s:Add(), 也不会有任何冲突, 那是它自己的s:Add()函数. 此外, 还可以定义一个全局的Add()函数, 同样区别于各个脚本中名为s:Add()的函数.

<SID>也可用于map. 它将使VIM为当前脚本生成一个ID, 这个ID将唯一标识当前脚本. 在我们的拼写校正的plugin例子中我们使用了这样的命令:

ex command

```
noremap <unique> <script> <Plug>TypecorrAdd  <SID>Add
..
noremap <SID>Add  :call <SID>Add(expand("<cword>"), 1)<CR>
```

当用户键入"\a"时, 将执行以下操作:

Display

```
a  ->  <Plug>TypecorrAdd  ->  <SID>Add  ->  :call <SID>Add()
```

如果另一个脚本中也映射了<SID>Add, 它将得到一个不同的脚本ID, 所以实际映射的内容也因之不同.

注意这里我们使用了<SID>Add()而不是s:Add(). 这是因为被映射的键将由最终用户键入, 所以应该在脚本外仍然生效. <SID>将被转换为一个脚本ID, VIM将据此决定在哪些脚本文件中查找Add()函数的定义.

这看起来真麻烦, 但要使多个plugin能和谐共存, 引入这样的复杂性是必需的. 基本的规则是: 在map中使用<SID>Add(), 而其它情况下使用s:Add()(脚本自己, autocommands, 用户自定义的命令)

也可以加入一个菜单项做同样的事情:

Display

```
noremenu <script> Plugin.Add Correction      <SID>Add
```

推荐在"Plugin"菜单下增加这样的菜单项. 此处只用到了一个菜单项, 要增加多个菜单项时, 最好是为它们创建一个子菜单. 比如, 要加入CVS相关操作的"Plugin.CVS.checkin", "Plugin.CVS.checkout", 可以加入一个名为"Plugin.CVS"的子菜单.

注意第28行的":noremap"命令，这可用于避免一个循环的或深层的map引入的问题．比如":call"可能已经被人映射为其它的某个东西．第24行也用到了":noremap"，但此处希望被映射的是"<SID>Add"． 所以使用了"<script>"关键字.它使这个mapping只局部有效于当前的脚本． |:map-script|，第26行使用":noremenu"的情况与此类似．

<SID> 和<Plug>

using-<Plug>

<SID>和<Plug>都是用于避免映射之间的混乱．注意两者的差别:

<Plug> 对脚本外部是可见的． 它用来定义一个键映射． <Plug>代表一个不可能键入的特殊代码． 为了最大限制地避免出现重复的键映射，最好使用这样的结构： <Plug> 脚本名映射名． 在此例中脚本名是"Typecorr"，映射名是"Add"． 结果就是"<Plug>TypecorrAdd"． 只有脚本名和映射名的第一个字符是大写的，这样可以看清楚映射名部分是从哪开始的．

<SID> 是脚本ID，脚本的唯一标识符．Vim在内部把<SID>转换为类型"<SNR>123_"的形式，其中"123"可以任何其它数字． 所以"<SID>Add()"可能在一个脚本中的是"<SNR>11_Add()"，在另一个脚本中就是"<SNR>22_Add()"．如果你用":function" 命令来列表系统中定义的函数时，你可能就会看到它． 映射中的<SID>与此完全一样，这样你可以在映射中调用一个脚本的局部函数．

用户命令

现在我们来添加一个用户定义命令来执行纠错:

code
```
if !exists(":Correct")
  command -nargs=1  Correct  :call s:Add(<q-args>, 0)
endif
```

用户自定义命令只有在没有同名命令存在的前提下才能定义．否则会得到一个错误．用":command!"命令覆盖原先的定义可不是好办法，这可能会让人疑惑为什么自己定义的命令现在不行了． 请参考|:command|

脚本变量

以"s:"开始的变量是脚本变量． 它只能在脚本内部使用． 对脚本外来说它是不可见的． 这样就避免了在不同脚本中使用同名变量的冲突．脚本变量在Vim运行其间将一直存在． 下次执行该脚本时它将保持原值．

有趣的是这些脚本变量还可以用在同一个脚本中定义的函数，自动命令和用户自定义命令中． 在上例中我们可以增加几行统计纠错的次数:

code
```
let s:count = 4
..
function s:Add(from, correct)
..
  let s:count = s:count + 1
  echo s:count . " corrections now"
endfunction
```

首先s:count变量在脚本中被初始化为4. 稍后调用s:Add()时，该变量增1. 从何处调用该函数并不重要，因为它的定义是在当前脚本中，所以它可以使用这些局部于脚本的变量.

结果

下面是完整的例子:

code

```
1 " Vim global plugin for correcting typing mistakes
2 " Last Change:  2000 Oct 15
3 " Maintainer:   Bram Moolenaar <Bram@vim.org>
4
5 if exists("loaded_typecorr")
6   finish
7 endif
8 let loaded_typecorr = 1
9
10 let s:save_cpo = &cpo
11 set cpo&vim
12
13 iabbrev teh the
14 iabbrev otehr other
15 iabbrev wnat want
16 iabbrev synchronisation
17         \ synchronization
18 let s:count = 4
19
20 if !hasmapto('<Plug>TypecorrAdd')
21   map <unique> <Leader>a  <Plug>TypecorrAdd
22 endif
23 noremap <unique> <script> <Plug>TypecorrAdd  <SID>Add
24
25 noremenu <script> Plugin.Add\ Correction      <SID>Add
26
27 noremap <SID>Add  :call <SID>Add(expand("<cword>"), 1)<CR>
28
29 function s:Add(from, correct)
30   let to = input("type the correction for " . a:from . ": ")
31   exe ":iabbrev " . a:from . " " . to
32   if a:correct | exe "normal viws\<C-R>\" \b\e" | endif
33   let s:count = s:count + 1
34   echo s:count . " corrections now"
35 endfunction
36
37 if !exists(":Correct")
38   command -nargs=1  Correct  :call s:Add(<q-args>, 0)
39 endif
40
41 let &cpo = s:save_cpo
```

这里面第33行还没有提到. 它的作用是对当前光标下的word进行拼写校正. 这里用|:normal|命令来执行新定义的缩写替换. 注意此处映射和缩写都进行了扩展, 即使是从一个以":noremap"定义的映射中进行函数

调用.

推荐将'fileformat'选项设为"unix". 这样Vim脚本在什么平台上都能正常工作. 将'fileformat'设为"dos"的脚本在Unix上就不行了. 参考|:source_crn|. 下面的命令可以确保文件格式设置正确:

shell command
```
:set fileformat=unix
```

文档

write-local-help

为你自己的plugin编写文档是个好习惯. 尤其是用户可以对你的plugin作出改变时. 请参考|add-local-help|了解如何安装帮助.

这里是一个简单的plugin帮助文件, 叫"typecorr.txt":

Display
```
*typecorr.txt*  Plugin for correcting typing mistakes

If you make typing mistakes, this plugin will have them corrected
automatically.

There are currently only a few corrections.  Add your own if you like.

Mappings:
<Leader>a   or   <Plug>TypecorrAdd
        Add a correction for the word under the cursor.

Commands:
:Correct word
        Add a correction for word.

*typecorr-settings*
This plugin doesn't have any settings.
```

整个文件中唯一必需遵循的格式是第一行. 该行内容将被取出来放在文件help.txt的"LOCAL ADDITIONS:"小节中, 参考|local-additions|. 第一行的第一列必需是字符"*".

你可以在你的帮助文件中定义很多标签, 放在两个**之路. 但注意不要与已有的帮助主题冲突. 最好在主题名中加入你的plugin本身的名字, 比如此例中的"typecorr-settings".

推荐你在自己的帮助文件中引用别的帮助主题时把它放在||中. 这会方便用户找到相关的帮助.

小结

plugin-special

使用plugin时注意事项总结:

s:name	局部于脚本的变量
<SID>	脚本ID，定义局部于脚本的映射和函数时使用.
hasmapto()	用来测试用户是否已经定义了实现某种功能的映射的函数
<Leader>	"mapleader"，用户定义的plugin映射名的起始符号.
:map <unique>	如果映射已经存在给出一个警告信息
:noremap <script>	使映射局部于脚本，对脚本以外不可见.
exists(":Cmd")	检查一个用户命令是否存在.

41.11 定制一个文件类型plugin *write-filetype-plugin*
ftplugin

文件类型plugin类似于全局的plugin，只不过它设置的选项和定义的映射只对当前缓冲区有效. 参考|add-filetype-plugin|了解如何使用这种plugin.

请先阅读|41.10|中关于全局plugin的部分. 其中的内容对文件类型plugin都同样有效. 不同的部分会在此特别指出. 最主要的一点是文件类型plugin只对当前缓冲区生效.

禁用

如果你在写一个可能会被很多人用到的plugin，别人就会需要一个禁用它的办法. 在你的plugin开头放上这样一段脚本:

code
```
" Only do this when not done yet for this buffer
if exists("b:did_ftplugin")
  finish
endif
let b:did_ftplugin = 1
```

这段脚本也兼有避免同一个plugin被执行多次的功能(比如使用":edit"而没有指定参数时)

现在用户可以用下面的命令来定义一个文件类型plugin来禁用你的plugin:

ex command
```
let b:did_ftplugin = 1
```

当然这需要他/她的plugin目录在'runtimepath'中出现中$VIMRUNTIME之前.

如果你想用默认的plugin，但又要改变它的部分设置，你可以在脚本中设置不同的选项:

ex command
```
setlocal textwidth=70
```

现在把脚本放在"after"目录下，这样它就会在标准发布的"vim.vim"之后被执行． 参考|after-directory|． 在Unix上该文件的位置为"~/.vim/after/ftplugin/vi
注意默认的plugin会设置"b:did_ftplugin"，但此处忽略了该变量．

撤消

undo_ftplugin

当用户执行了":setfiletype xyz"时前一次设置文件类型的效果就撤消了． 可以把"b:undo_ftplugin"变量赋予为撤消设置的命令． 如下：

ex command
```
let b:undo_ftplugin = "setlocal fo< com< tw< commentstring<"
        \ . "| unlet b:match_ignorecase b:match_words b:match_skip"
```

使用":setlocal"然后在选项名后面跟上"<"将会把选项重置为全局设置值．这可能是重置选项的最佳策略．

上面的例子要求选项'cpoptions'中不包括"C"项，以支持跨行命令，参考|use-cpo-save|．

文件名

文件的类型名必需包含在plugin文件名中，参考|ftplugin-name|．可以采用下面的3种形式之一：

List
```
.../ftplugin/stuff.vim
.../ftplugin/stuff_foo.vim
.../ftplugin/stuff/bar.vim
```

"stuff" 是文件类型名，"foo"和"bar"可以是任意的名字．．

选项

要让文件类型plugin只影响当前的缓冲区可以用

ex command
```
:setlocal
```

命令来设置选项．同时只设置那些局部于缓冲区的选项(请参考关于选项的帮助检查一个选项是否局部于缓冲区)． 用|:setlocal|命令来设置一个

全局选项或局部于窗口的选项时，作出的改变会同时影响多个缓冲区，这可不是一个文件类型plugin应该做的事.

如果选项的类型是标志和可选项的集合，最好用"+="和"-="来保留原值. 注意用户自己可能已经改变了该选项. 所以先把选项设为默认值再进行调整会更好. 比如:

ex command
```
:setlocal formatoptions& formatoptions+=ro
```

映射

要使映射只对当前缓冲区生效，可以用

ex command
```
:map <buffer>
```

命令. 这需要用到上面提到的两步映射法. 下面是在一个文件类型plugin中定义某个功能的例子:

code
```
if !hasmapto('<Plug>JavaImport')
  map <buffer> <unique> <LocalLeader>i <Plug>JavaImport
endif
noremap <buffer> <unique> <Plug>JavaImport oimport ""<Left><Esc>
```

|hasmapto()|用来检查是否用户已经定义了一个映射来执行<Plug>JavaImport. 如果没有，则定义默认的映射. 以|<LocalLeader>|开始，这样可以让用户决定映射的起始符号. 默认是反斜杠. "<unique>"使映射已经定义或者与一个既有映射功能重叠时给出错误消息. |:noremap|用于避免其它映射的介入. 你可能希望用":noremap <script>"来避免在脚本中重定义以<SID>开头的映射.

用户必需有机会禁用在文件类型plugin中定义的映射，同时又不伤及其它功能. 下面是一个邮件plugin中实现这一目的的例子:

code
```
" Add mappings, unless the user didn't want this.
if !exists("no_plugin_maps") && !exists("no_mail_maps")
  " Quote text by inserting "> "
  if !hasmapto('<Plug>MailQuote')
    vmap <buffer> <LocalLeader>q <Plug>MailQuote
    nmap <buffer> <LocalLeader>q <Plug>MailQuote
  endif
  vnoremap <buffer> <Plug>MailQuote :s/^/> /<CR>
  nnoremap <buffer> <Plug>MailQuote :.,$s/^/> /<CR>
endif
```

这里用到了两个全局变量: no_plugin_maps 禁用所有文件类型plugins中的映射
no_mail_maps 禁用邮件plugin中的映射

用户命令

要为某种特定类型的文件加一个命令，并使该命令只能在当前缓冲区中使用. 可以为|:command|命令加"-buffer"参数. 如[1]:

ex command
```
:command -buffer  Make  make \%:r.s
```

变量

Vim会对缓冲区执行相应的plugin脚本. 脚本局部变量|s:var|会在脚本的多次执行中被共享. 如果你想让某个变量局部于缓冲那就代而使用缓冲区局部变量.

函数

函数只需定义一次. 但文件类型plugin却要在每个缓冲区符合其目标文件类型时都被执行. 下面的技巧可以避免函数的多次定义:

code
```
:if !exists("*s:Func")
:  function s:Func(arg)
:    ...
:  endfunction
:endif
```

小结

ftplugin-special

使用文件类型plugin的注意事项小结:

<LocalLeader>	"maplocalleader", 用户定义的文件类型plugin中定义的映射的起始符号.
:map <buffer>	定义局部于缓冲区的映射
:noremap <script>	只重映射在当前脚本中以<SID>起始的映射
:setlocal	只对当前缓冲区设置选项
:command -buffer	定义一个局部于当前缓冲区的命令
exists("*s:Func")	检查一个函数是否定义

请参考|plugin-special|了解适用于所有的plugin的技巧.

[1]译: 此处的one buffer有歧义

41.12 定制一个编译器plugin *write-compiler-plugin*

一个编译器plugin为某种特定的编译器设置选项. 用户可以用|:compiler|命令载入它. 它最大的用途就是设置'errorformat'和'makeprg'两个选项.

最好先看一个例子, 下面的命令将会编辑所有默认的编译器plugin:

ex command
```
:next $VIMRUNTIME/compiler/*.vim
```

使用|:next|命令到下一个plugin文件.

这些文件有两个特别之处. 其一是它提供了一种机制允许用户覆盖缺省的文件或向缺省文件增加设置. 缺省文件一般这样开头:

code
```
:if exists("current_compiler")
:  finish
:endif
:let current_compiler = "mine"
```

一旦你写了自己的编译器plugin并把它放到了自己的runtime目录(比如在Unix上是~/.vim/compiler), 你就可以在其中设置"current_compiler"变量以避免缺省的编译器plugin设置.

第二是用":set"设置":compiler!", 用":setlocal" 设置":compiler". Vim定义了一个用户命令":CompilerSet"来进行这一设置, 不过老版的Vim中没有, 这时你就要在自己的plugin中定义了, 如下:

ex command
```
if exists(":CompilerSet") != 2
  command -nargs=* CompilerSet setlocal <args>
endif
CompilerSet errorformat&                " use the default 'errorformat'
CompilerSet makeprg=nmake
```

如果你要写一个供整个系统使用或放到Vim发行版中的编译器plugin, 最好使用上面示范的机制. 这样用户就可以通过"current_compiler"变量来决定对该plugin的取舍了.

如果你要写的编译器plugin目的是覆盖掉缺省plugin的设置, 那就不要检查"current_compiler". 这样的plugin被系统假设是最后执行的, 所以它所在的目录应该位于'runtimepath'的末尾. 对Unix系统来说很可能是~/.vim/runtime/after/compiler.

下一章：	|usr_42.txt| 增加新菜单
版　权：	请参考|manual-copyright| vim:tw=78:ts=8:ft=help:norl:

usr_42.txt Vim 6.3版.最后修改: 2002年10月08日

VIM用户手册--- 作者: Bram Moolenaar

增加新菜单

现在你应该知道Vim是何其灵活，这包括在GUI环境下对菜单的使用，你可以通过定义自己的菜单项来更方便地使用一些命令. 当然，这都是对善用鼠标的用户而言.

下一章: |usr_43.txt| 使用filetypes
前一章: |usr_41.txt| Vim脚本
目 录: |usr_toc.txt|

42.1 介绍

vim的菜单在"$VIMRUNTIME/menu.vim"中定义，在定义你自己的菜单之前，最好先看一下这个文件. 要定义一个菜单项，使用":menu"命令. 其基本语法如下:

ex command
```
:menu {menu-item} {keys}
```

{menu-item}描述了菜单项要放的位置. 一个典型的{menu-item}如"File.Save"，代表名为"File"的菜单下的"Save"菜单项. 点号用于分隔菜单的名字和其下的菜单项的名字. 如下:

ex command
```
:menu File.Save :update<CR>
```

":update"命令在文件被改写时保存文件.

你还可以定义级联菜单: "Edit.Settings.Shiftwidth"定义了菜单"Edit"下的一个子菜单"Settings"和该子菜单的菜单项"Shiftwidth". 你还可以定义更深层的级联菜单,不过最好不要使用太多层的子菜单，因为你得不停地移动鼠标.

":menu"命令很类似":map"命令： 它们都是以左边定义如何触发一个操作，右边定义操作的具体内容．{keys}指定的就是你的种种模式下实际要键入的字符．所以在插入模式下，如果指定的{keys}是普通文本，那么执行这一操作就会在当前光标处插入这些文本．

加速键

符号&将定义它后面的字符为一个加速键．例如，按下ALT-F可以选择"File"菜单，再按下S可以选择"Save"菜单项(通过'winaltkyes'选项可以禁用这一功能)．对应的{menu-item} 就是"&File.&Save"． 加速键字符在菜单中显示时有一个下划线．必需注意每个加速键字符中同一级平行菜单中只能定义一次． 否则的话触发该加速键到底是执行哪个操作呢．Vim不会对此发出警告，自己小心!

优先级

File.Save 菜单项的实际定义如下：

ex command
```
:menu 10.340 &File.&Save<Tab>:w  :confirm w<CR>
```

其中数字10.340 是优先号．Vim据此来决定把菜单项放在什么位置．第一个数字(10)指定菜单在菜单栏上的位置．小数靠左，大数靠右．

下面是标准菜单的优先级定义：

Display
```
 10     20     40      50      60       70         9999

+-------------------------------------------------------+
| File   Edit   Tools   Syntax   Buffers   Window      Help |
+-------------------------------------------------------+
```

Help菜单的优先级被定义为一个很大的数字，这样可以使它总是处于菜单栏的最右边．

第二个数字(340)决定了菜单项在下拉菜单中的位置．小的靠上，大的居下． 下面是File菜单中的优先级定义：

Display
```
                    +-----------------+
10.310              |Open...          |
10.320              |Split-Open...    |
10.325              |New              |
10.330              |Close            |
10.335              |---------------- |
10.340              |Save             |
10.350              |Save As...       |
10.400              |---------------- |
10.410              |Split Diff with  |
10.420              |Split Patched By |
10.500              |---------------- |
10.510              |Print            |
10.600              |---------------- |
10.610              |Save-Exit        |
10.620              |Exit             |
                    +-----------------+
```

你也许已经注意到这些优先级数字并不是一个紧挨一个．这是为你向中间插入自己的菜单项留下可能.(最好还是不要动这些标准菜单，另外定义一个新的菜单去加入你的菜单项)

如果你定义了子菜单． 还可以为优先码附加一个对应的".数字".

特殊字符的考虑

本例中的{menu-item}是"&File.&Save<Tab>:w"． 这揭示了很重要的一点： {menu-item}必需是一个字． 如果你想在其中放入点号，空格或跳格健就得用别的办法： 要么使用尖括号表示法(比如<space>和<tab>)，要么用反斜杠表示的脱字符序列：

ex command
```
:menu 10.305 &File.&Do It... :exit<CR>
```

上例中，名为"Do It..."的菜单项就包含了一个空格，对应的命令是":exit<CR>".

菜单项中的<Tab>用于分隔菜单项的名字和用户提示． <Tab>后面的部分显示在菜单项的最右边． 在File.Save菜单项中形如"&File.Save<Tab>:w"．其中菜单名为"File.Save"，提示语是":w".

分隔线

分隔线用于在视觉上把相关的菜单项划为一组，定义分隔线时菜单项名字的开头和结尾都必需是字符"-"． 如"-sep-"． 在同一菜单中使用多个分隔线时这些名字必需是唯一的．

为分隔线定义的命令永远不会被执行，但你还不能省略它． 放一个冒号就可以了.如:

ex command
```
:amenu 20.510 Edit.-sep3- :
```

42.2 菜单命令

你可以定义一些只在特定模式才出现的菜单项，就象":map"命令的变体：

List
```
:menu           Normal, Visual and Operator-pending mode
:nmenu          Normal mode
:vmenu          Visual mode
:omenu          Operator-pending mode
:menu!          Insert and Command-line mode
:imenu          Insert mode
:cmenu          Command-line mode
:amenu          All modes
```

为避免定义在菜单项中的命令已经被":map"映射，使用形如":noremenu"，":nnoremenu"，":anoremenu"的命令．

使用:AMENU

":amenu"命令有点特殊．它假定定义的{keys}在普通模式下被执行．如果当前模式是Visual或插入模式，Vim执行操作前会回到Normal模式．":amenu"命令自动为你插入CTRL-C 或CTRL-O．比如下面的命令：

ex command
```
:amenu  90.100 Mine.Find Word  *
```

各种模式下的结果如下：

List
```
Normal mode:              *
Visual mode:              CTRL-C *
Operator-pending mode:    CTRL-C *
Insert mode:              CTRL-O *
Command-line mode:        CTRL-C *
```

在命令行模式下CTRL-C会放弃当前当前已经键入的内容．在Visual和等待操作模式下CTRL-C将终止该模式． 而在插入模式下的CTRL-O则会临时进入Normal模式，执行完该命令后再返回插入模式．

因为CTRL-O只能为一个命令在插入模式下开绿灯. 所以如果你要执行多个Normal模式下的命令, 就要借助函数, 如下:

code
```
:amenu  Mine.Next File  :call <SID>NextFile()<CR>
:function <SID>NextFile()
:  next
:  1/^Code
:endfunction
```

执行这个菜单命令将跳转到下一个文件":next". 然后搜索以"Code"开始的文本行.

函数名前的<SID>是脚本ID. 以此定义的函数局部于当前的Vim脚本. 这样就避免了与其它脚本文件中定义的同名函数发生冲突. 参见|<SID>|.

无回显菜单

菜单执行{kyes}中的命令与你实际键入这些命令一样. 对于一个":"命令来说命令本身会被回显在命令行, 如果命令行很长, 还会出现'请按ENTER 或其它命令继续', 有时候这可真是烦人. 为避免这种情况, 可以通过<silent>参数把菜单定义为无回显的. 比如, 调用上例中的NextFile()函数, 你会在命令行上看到:

ex command
```
:call <SNR>34_NextFile()
```

在定义菜单时插入一个<silent>作为命令的第一个参数就可以避免看到它:

ex command
```
:amenu <silent> Mine.Next File :call <SID>NextFile()<CR>
```

但也不要到处去用"<silent>". 对于短命令来说没有必要. 如果的自定义菜单还有别的用户. 能让他在执行菜单操作时看到实际进行的操作是一个不错的回馈.

菜单列表

使用一个没有定义{keys}的菜单命令, 将会列出该菜单的定义(如果该菜单已定义的话). 你可以指定{menu-item}或它的一部分来列出某个菜单. 如:

ex command
```
:amenu
```

该命令列出所有的菜单. 那可是一份很长的清单! 给出要显示的菜单名会让显示列表更集中也更短一些:

ex command
```
:amenu Edit
```

这就只会列出"Edit"菜单在各种模式下的定义了. 要列出某个菜单中插入模式下的定义, 使用命令:

ex command
```
:imenu Edit.Undo
```

注意要一字不差地键入菜单名. 大小写也不能含糊. 但是定义加速键的"&"可以省略. <Tab>和它后面的部分也不用输入.

删除菜单

要删除一个菜单, 使用与显示菜单同样的命令, 只是命令名从"menu"变为"unmenu". 这样一来, ":menu"变成":unmenu", "nmenu"变成"nunmenu", 如此等等. 如要删除插入模式下的"Tools.Make"菜单项:

ex command
```
:iunmenu Tools.Make
```

也可以删除整个菜单, 只指定菜单名: 如:

ex command
```
:aunmenu Syntax
```

该命令删除Syntax菜单和它的所有子项.

42.3 其它

你可以通过'guioptions'选项来改变菜单的外观. 它的默认设置包括了所有标志值. 你可以用如下命令来去除某个标志值:

List
```
:set guioptions-=m
m       移除该选项菜单条就不会显示出来了.
M       移除该标志默认菜单将不会被加载
g       移除该标志将使不可用的菜单项被移除而不仅仅是变灰. (很多
        系统上该功能都不能用)
t       移除该标志使菜单成为浮点窗口的功能将失效.
```

每个菜单列表顶端的虚线可不是分隔线. 当你选择使用该命令时, 菜单将好象被从该虚线处剪下一样, 变成浮动窗口. 这叫剪贴菜单(tearoff, 金山词霸里没有这个词, 倒是有tear-off, 解释为"可按虚线剪下的纸"). 当你要频繁使用同一菜单时这可让你用起来更方便.

要翻译菜单项, 参见|:menutrans|.

因为使用菜单时要用到鼠标, 所以最好用":browse"命令去选择文件. 用":confirm"去打开一个对话框, 而不是去看命令行上的出错信息. 当前缓冲区被改变时, 可以组合使用这两个命令:

ex command
```
:amenu File.Open  :browse confirm edit<CR>
```

":browse"命令会打开一个文件浏览窗口用于选择一个待编辑的文件. ":confirm"将在当前缓冲区被改变时弹出一个确认对话框. 你可以选择保存改变或放弃或撤消操作.

要进行更多的控制, 可以使用confirm()和inputdialog()函数. 默认菜单里面就有一些这样的例子.

42.4 工具栏和弹出式菜单

有两种菜单情况特殊: 工具栏和弹出式菜单. 以此起始的菜单名将不会出现在通常的菜单条里.

工具栏

工具栏只有在'guioptions'选项里包括"T"标志时才会出现.

工具栏使用图标而不是文字表示命令. 比如, 名为"ToolBar.New"的{menu-item}将在工具栏上定义一个"New"图标.

Vim编辑器内置了28个图标. 你可以在|builtin-tools|处找到它们. 很多图标都被用于默认的工具栏. 你也可以重新定义这些图标的操作.

你可以为某个工具项另外指定一个位图加到工具栏上. 或者以一个新的图示来定义一个新的工具项. 例如下面的命令定义了一个新的工具项:

ex command
```
:tmenu ToolBar.Compile  Compile the current file
:amenu ToolBar.Compile  :!cc % -o %:r<CR>
```

现在你需要创建一个图标. 对MS-Windows而言必需是bitmap格式, 文件名为"Compile.bmp". 对Unix来说用XPM格式, 文件名为"Compile.xpm".[1].

[1] 译注: 当然MS-Windows还是大小写不敏感的, 而Unix是大小写敏感的

图标尺寸也必需是18 x 18像素. 在MS-Windows上也可以用其它尺寸的图标, 但看起来就不那么顺眼了.

把你创建的bitmap放在'runtimepath'选项指定的任何一个目录下名为"bitmaps"的子目录. 如在Unix系统上"~/.vim/bitmaps/Compile.xpm".

你可以为工具栏上的工具项定义一个提示语. 一个提示语就是告诉用户这些工具能做什么的简短描述. 比如"Open file". 鼠标位于工具项上面时将会显示这些提示语. 如果图标并不能使人望文生义, 提供这样一个语言描述就十分有用了:

ex command
```
:tmenu ToolBar.Make  Run make in the current directory
```

> 备注: 留心大小写. "Toolbar"和"toolbar"可都不是"ToolBar"!

删除一个提示, 用|:tunmenu|命令.

'toolbar'选项可用于控制是显示图标还是文字, 还是兼而有之. 大多数人喜欢只用图标. 因为文字会占去显示器上很大地方.

弹出菜单

弹出菜单在鼠标所在位置处弹出. 在MS-Windows上可以通过右击鼠标来激活它. 然后用左健你可以选择它上面的菜单项. 在Unix上弹出菜单通过按下右健, 移动到要用的菜单项, 然后松开右健使用[1].

弹出菜单只有在选项'mousemodel'被设置为'popup'或'popup_setpos'时才会出现. 这两个选项的不同在于"popup_setpos"会把光标移动到鼠标的当前位置. 在一个被选择区域内点击时, 选择区域保持不变. 而在选择区域之外单击时, 则会使区域选择被撤消.

每种模式下都有各自的弹出菜单. 所以弹出菜单不会象常规菜单那样出现不可用的灰色菜单项.

下一章:	|usr_43.txt| 文件类型
版 权:	请参考|manual-copyright| vim:tw=78:ts=4:ft=help:norl:

[1]译注: 有机会用Xwindows时你自然会明白

usr_43.txt Vim 6.3版.最后修改: 2002年07月14日

VIM用户手册--- 作者: Bram Moolenaar

文件类型

当你编辑一种类型的文件时，比如C程序或shell脚本，你经常会使用一组相同的选项设置和键映射. 每次都设置这些相同的东西很快会让人厌烦. 本章教你如何让它自动化.

下一章: |usr_44.txt| 定制自己的语法高亮
前一章: |usr_42.txt| 添加新的菜单
目　录: |usr_toc.txt|

43.1 文件类型的插件 *filetype-plugin*

如何使用文件类型插件已经在|add-filetype-plugin|帮助主题中讨论过了. 但你可能不会满意于默认的设置，因为它们为了兼容性的原因被削减为一个最小可用的集合. 假设你要在C源文件中设置'softtabstop'选项为4并且定义一个映射来插入一个三行式注释. 以下二步会满意你的要求:

your-runtime-dir

1. 创建你自己的运行时目录. 在Unix系统上通常是"~/.vim". 在该目录下创建名为"ftplugin"[1]的目录:

shell command
```
mkdir ~/.vim
mkdir ~/.vim/ftplugin
```

如果你不用Unix，看一看'runtimepath'选项的设置，找出Vim会在哪里寻找"ftplugin"目录:

ex command
```
set runtimepath
```

[1]译注:是ft+plugin，而不是ftp + lugin

一般人会用第一个目录(第一个逗号之前)，但是如果你不想接受默认目录的话可以在你的|vimrc|文件里添加一个目录到'runtimepath'选项上去.

2. 创建文件"~/.vim/ftplugin/c.vim"，内容如下:

ex command
```
setlocal softtabstop=4
noremap <buffer> <LocalLeader>c o/**************<CR><CR>/<Esc>
```

编辑一个C源文件试试，你会看到'softtabstop'选项已经被设置为4[1]. 但如果你转而去编辑另一个文件它又会被重置为零.因为这里使用的是":setlocal"命令. 该命令使'softtabstop'选项的值局部于当前缓冲区. 一旦切换到别的缓冲区，它就会被重置为那个缓冲区的值. 对于新的缓冲区来说会使用默认值或者是最后一次通过":set"命令设置的值.

同样，对于"\c"的映射键会在进入另一个缓冲区时消失. ":map <buffer>"命令会创建一个局部于当前缓冲区的映射键. 所有的映射命令都是如此: ":map!"，":vmap"，等等映射键中的|<LocalLeader>|会被"maplocalleader"替换.

你可以在下面这个目录中发现一个参考例子:

List
```
$VIMRUNTIME/ftplugin/
```

关于写filetype 插件的更多信息，参见: |write-plugin|.

43.2 添加一个filetype

如果你正在使用一种Vim还不认识的文件格式[2]，本节内容将教你如何让Vim识别它. 你需要一个自己的运行时目录，参见上面的|your-runtime-dir|.

创建一个名为"filetype.vim"的文件，加入你自己的filetype相关的autocommand. (Autocommands 在|40.3|一节中有详细解释.) 如:

ex command
```
augroup filetypedetect
au BufNewFile,BufRead *.xyz     setf xyz
augroup END
```

Vim将据此把所有以".xyz"为扩展名的文件标识为"xyz"文件类型. ":augroup"命令把这个autocommand放在"filetypedetect"组. 这样，":filetype off"将会移除所有关于文件类型自动检测的autocommand. "setf"命令会

[1]译注: 用命令:set softtabstop可以看到
[2]译注: 如果真是这样，要么你在定义自己的文件类型，要么，你就是一个计算机语言专家或文件类型专家

把'filetype'设置为相应的值,除非当前缓冲区的'filetype'已经设置好了. 这样也确保了'filetype'不会被设置两次.

你可以使用众多的模式符来确定你的文件类型. 其中也包括目录名匹配检查.参见|autocmd-patterns|. 如,"/usr/share/scripts/"目录下全是"ruby"文件,但是没有正常的文件扩展名,加入下面的命令会正确设置你的filetype:

ex command
```
augroup filetypedetect
au BufNewFile,BufRead *.xyz                    setf xyz
au BufNewFile,BufRead /usr/share/scripts/*     setf ruby
augroup END
```

但是,如果你正在编辑/usr/share/scripts/README.txt,当然从文件名判断它不是一个ruby文件. 但是问题在于一个以"*"为结尾的模式会匹配所有文件. 为避免这个问题,可以把filetype.vim放在'runtimepath'选项中最后一个目录中. 对Unix用户来说,你可以放在"~/.vim/after/filetype.vim".

现在你可以在~/.vim/filetype.vim中放上你的文本文件检测了:

ex command
```
augroup filetypedetect
au BufNewFile,BufRead *.txt                    setf text
augroup END
```

文件"filetype.vim"会先在'runtimepath'选项中的路径里被发现. 接下来执行最后发现的~/.vim/after/filetype.vim:

ex command
```
augroup filetypedetect
au BufNewFile,BufRead /usr/share/scripts/*     setf ruby
augroup END
```

现在的情况是Vim在'runtimepath'的每个目录中搜索"filetype.vim",先是找到了~/.vim/filetype.vim. 检查*.txt的autocommand在那里捕获每一个扩展名为.txt 的文件,设置filetype为text,然后Vim在作为'runtimepath'子集的$VIMRUNTIME里发现了~/.vim/after/filetype.vim,加上检测/usr/share/scripts/下脚本的命令.

现在你再去打开/usr/share/scripts/README.txt,autocommand的执行以上面的顺序进行: 找到"*.txt"模式,执行"setf text"将文件类型设置为"text". ruby对应的模式也被匹配了. 再执行"setf ruby",但此时'filetype'已经被设为了"text",所以此处的"setf ruby"不再生效.

你若是编辑文件/usr/share/scripts/foobar,Vim以同样的顺序执行匹配的autocommand命令. 只有符合ruby模式定义的缓冲区其'filetype'才会被设置为ruby.

根据内容识别

如果你的文件不能通过名字来识别，还可以通过文件的内容来确定文件类型，比如，多数脚本文件以此作为第一行：

Display
```
#!/bin/xyz
```

为识别此类文件,在你的运行时目录(跟filetype.vim一样)里创建一个名为"scripts.vim"的文件,内容类似于：

code
```
if did_filetype()
  finish
endif
if getline(1) =~ '^#!.*[/\\]xyz\>'
  setf xyz
endif
```

第一道检查是did_filetype()，这是为了避免那些文件类型已经通过filetype正确识别的缓冲区在此浪费时间.

scripts.vim文件的执行通过在filetype.vim文件里的一个autocommand定义被触发. 所以，整个识别过程的顺序是：

1. 在'runtimepath'中位于$VIMRUNTIME之前的filetype.vim文件
2. $VIMRUNTIME/filetype.vim的第一部分
3. 'runtimepath'目录中所有名为scripts.vim的文件
4. $VIMRUNTIME/filetype.vim的剩余部分
5. 在'runtimepath'中位于$VIMRUNTIME之后的filetype.vim文件

如果这些还不合你用，那就增加一个能匹配所有文件的autocommand，执行一个脚本或函数去检查你的文件内容吧.

下一章：|usr_44.txt| 自定义语法高亮文件

usr_44.txt Vim 6.3版.最后修改: 2002年10月10日

VIM用户手册--- 作者: Bram Moolenaar

自定义语法高亮

Vim拥有对数百种文件类型进行语法高亮的功能. 如果你要编辑的文件仍不在其中，本章将会带你发现如何为你的新文件类型定制语法高亮. 建议参考|syn-define|.

下一章: |usr_45.txt| 选择你的语言
前一章: |usr_43.txt| 使用文件类型
目 录: |usr_toc.txt|

44.1 基本语法命令

从一个已有的语法文件开始会节省你大量时间. 你可以在$VIMRUNTIME/syntax目录下找到一个近似于你的新语言的语法文件. 这些文件同时也向你揭示了一个语法文件的通常架构. 不过你还需要继续下面的内容来理解它.

从基本的开始，在我们定义一种新的语法规则之前，首先要做是就是清除已定义的旧规则:

ex command
```
:syntax clear
```

这对于最终的语法文件来说并不是必需的，但在实验这些功能时还是十分有用.

本章内容已经是大大简化． 如果你要写一个为他人使用的语法文件，那可要从头到尾好好通读所有与此相关的细节了．

列出已定义的语法项

要巡视一下当前已经定义了哪些语法项，使用命令：

ex command
```
:syntax
```

这个命令会为你检查当前实际定义了哪些语法项．对于正在尝试定制一个新语法文件的你是十分有用的． 该命令也会显示每个语法项的颜色定义，让你分清楚当前文件中的以各种颜色显示的文本都对应到哪些语法项．

要列出一个指定的语法组中的语法项，使用命令：

ex command
```
:syntax list {group-name}
```

在前面加上一个@号，该命令也可用于列出聚簇中的语法定义(在|44.8|节中有解释)

大小写敏感性

一些语言是大小写不敏感的，比如Pascal． 而其它语言如C则是大小写敏感的． 下面的命令决定了语法规则的匹配是否是大小写敏感的：

ex command
```
:syntax case match
:syntax case ignore
```

参数"match"意味着Vim在匹配语法元素时是大小写敏感的． 如此一来，"int"跟"Int"和"INT"就都是不同的东西了． 如果用的参数是"ignore"，"Procedure"，"PROCEDURE"，以及"procedure"就都被视为相同的语法元素了．

":syntax case"命令可以出现在一个语法文件的任意位置并影响该位置以后的语法定义．多数情况下，一个语法文件中只需要一个":syntax case"命令；不过如果你使用一种有时大小写敏感有时又不敏感的非常规语言，也可以在整个文件中使用多个":syntax case"命令．

44.2 关键字

最基本的语法元素就是关键字． 下面的命令定义一个关键字：

ex command
```
:syntax keyword {group} {keyword} ...
```

{group}是语法组的名字. 使用":highlight"命令你可以将一组颜色方案应用到该{group} 语法组上. {keyword}参数指定了实际的关键字,下面是一个例子:

ex command
```
:syntax keyword xType int long char
:syntax keyword xStatement if then else endif
```

本例中使用了名为"xType"和"xStatement"的语法组. 根据约定,每个组名都前辍以该语言的filetype. 本例中定义了x语言(无趣的eXample语言). 在"csh"脚本的语言文件中组名应该是"cshType". 即组名的前辍就是'filetype'选项的值.

这些命令将使"int","long"和"char"以同一种方式高亮显示,而"then","else" 和"endif"以另一种方式. 现在可以把定义好的x组名与标准的Vim组名联系起来:

ex command
```
:highlight link xType Type
:highlight link xStatement Statement
```

该命令使Vim对组"xType"应用与"Type"相同的语法高亮,对"xStatement"应用与"Statement"相同的语法高亮. |group-name|主题中有标准组名的信息.

生僻的关键字

定义的关键字必需是'iskeyword'选项的定义. 如果你用到了额外的字符,该关键字将不会被匹配到,Vim也不会就此给出错误信息.

若x语言要在关键字中使用"-"字符,可以这样做:

ex command
```
:setlocal iskeyword+=-
:syntax keyword xStatement when-not
```

":setlocal"命令将使对"iskeyword"的改变只对当前缓冲区有效. 同时它也改变了"w"和"*"命令的行为. 如果你不希望这样,那就不要用关键字,用match命令(见下一节).

假设x语言允许缩写. 比如"next"可以被缩写为"n","ne"或"nex". 这样的规则可以用下面的命令定义:

ex command
```
:syntax keyword xStatement n[ext]
```

放心，"nextone"不会被匹配，关键字只适用于一个完整的词.

44.3 匹配

考虑一下如何定义一个复杂一点的语法项. 比如要匹配一个标识符. 要做到这一点，需要使用match语法. 下面的命令将匹配所有以小写字母组成的词:

ex command
```
:syntax match xIdentifier /\<\l\+\>/
```

> 备注: 关键字会凌驾于任何其它的语法项定义. 所以"if", "then"等等将被认为是关键字，即使它们同时也符合上例中的xIdentifier.

上例中命令的最后一部分是一个模式，正如在搜索文本时的用法. //用于界定一个模式(就象在:substitute命令中那样). 也可以用/之外的其它字符，如+号或"号.

现在来定义一个注释. 在x语言中假设注释是自#至行尾的内容:

ex command
```
:syntax match xComment /#.*/
```

因为这里可以使用模式匹配，所以可以以此匹配非常复杂的语法项. 参见|pattern|可以了解更多关于查找模式的问题.

44.4 区域

在x语言中，字符串定义为由两个双引号包围起来的字符序列. 要高亮一个字符串需要定义一个区域. 该区域以双引号开始，同样以双引号结尾. 定义如下:

ex command
```
:syntax region xString start=/"/ end=/"/
```

"start"和"end"分别定义了该区域的开始和结尾. 考虑下面这句将被如何匹配:

ex command
```
"A string with a double quote (\") in it"
```

这引出了一个问题：在一个字符串中间的双引号将结束整个字符串的识别．所以还需要告诉Vim跳过这些以\"形式表达的脱字符．这要在命令中用到skip关键字：

ex command
```
:syntax region xString start=/"/ skip=/\\"/ end=/"/
```

两个反斜杠\\匹配到一个真正的反斜杠\字符，因为反斜杠\在模式中是一个特殊字符．

何时用region而不用match呢？两者的主要不同在于match是一个完整的模式,一次匹配一整个字符序列．　而一个区域以"start"指定的模式为开始，以"end" 指定的模式为结束．　区域中的"end"模式可能被匹配也可能不被匹配．　所以如果你要定义一个一定以某个模式为结束的语法项．那就不能用region来定义．另外，区域定义看起来更简单.而且方便定义嵌套的语法项．嵌套语法项在下节讲述．

44.5　嵌套语法项

看一下下面的注释：

Display
```
%Get input  TODO: Skip white space
```

现在假设要让TODO以黄色来高亮显示，尽管它已经包含在一个定义为蓝色高亮的注释中．　要让Vim识别这种情况，可以定义下面的语法组：

ex command
```
:syntax keyword xTodo TODO contained
:syntax match xComment /%.*/ contains=xTodo
```

第一行中"contained"参数告诉Vim该关键字只能存在于另一个语法项中．　下一行的"contains=xTodo"则说明允许一个组名xTodo的语法元素嵌套在其中．　结果是整个注释行匹配到"xComment"组被显示为蓝色．而其中的TODO匹配到xTodo组而被显示为黄色．

递归嵌套

x语言定义了以花括号{}括起来的部分为一个代码块．　当然一个代码可以包含另一个代码块．　这样就需要下面的定义：

ex command
```
:syntax region xBlock start=/{/ end=/}/ contains=xBlock
```

假设有这样的x代码：

code
```
        while i < b {
                if a {
                        b = c;
                }
        }
```

首先一个xBlock组匹配到第一行的{. 第二行又发现了一个{. 因为这此处已经位于外围的xBlock内. 所以"b = c"这一行就位于第二级的xBlock区域里, 接下来的一行里又发现了}字符. 它符合xBlock区域的结束符定义. 因此它结束了嵌套的内层xBlock. 由于}匹配的是内层的xBlock, 所以外层的xBlock区域将视之为普通文本. 而下一行的}才匹配到第一个xBlock区域的结束.

保留行尾

考虑下面的语法项定义:

ex command
```
:syntax region xComment start=/%/ end=/$/ contained
:syntax region xPreProc start=/#/ end=/$/ contains=xComment
```

这里定义了一个注释: 以%开始直到行尾. 一个预处理指示符, 以#开始直到行尾. 因为一个注释也可以出现在一个预处理定义中.所以预处理项的定义包含了一个"contains=xComment"参数. 现在来看看将它应用于下面的文本会发生什么:

code
```
        #define X = Y  % Comment text
        int foo = 1;
```

你会看到第二行也被以xPreProc被高亮. 预处理项应该在行尾就结束了. 要不干吗要定义"end=/$/", 到底哪出了问题?

问题在于被包含的注释. 注释项以%开始直到行尾. 所以注释项的匹配结束后, 预处理项的语法还没有结束. 而仅有的一个行尾已经因为符合注释项的定义而被吃掉了. 所以预处理项的匹配结束于下一行的行尾, 这就是为什么第二行被识别为预处理项的一部分.

为避免一个被包含的语法项吃掉行尾, 要在命令中加一个额外的"keepend"参数. 这个参数可以使Vim正确处理需要匹配到行尾两次的语法项.

ex command
```
 :syntax region xComment start=/%/ end=/$/ contained
 :syntax region xPreProc start=/#/ end=/$/ contains=xComment keepend
```

嵌套多个语法项

contains参数还可以指定一个语法项可以包含其它任何的语法项，如：

ex command
```
:syntax region xList start=/\[/ end=/\]/ contains=ALL
```

这使xList可以包含任何语法项，包括它自己，但是不是原位置的同一个xList(这样会引起死循环).

你还可以指定某个语法项被排除在外. 这样可以定义除某个语法项之外的所有语法项：

ex command
```
:syntax region xList start=/\[/ end=/\]/ contains=ALLBUT,xString
```

如果用了"contains=TOP"，该语法项就可以包含所有没有"contained"参数的其它语法项. "contains=CONTAINED"则用于定义该语法项只包含那些有"contained"参数的语法项.参见|:syn-contains|了解更多的细节.

44.6 后续组

x语言有如下形式的语句：

code
```
        if (condition) then
```

假设你要以不同的规则来高亮这三项. 但是"(condition)"和"then"也可能在别处出现，而在那里它们又以不同的规则来高亮. 看下面的命令：

ex command
```
:syntax match xIf /if/ nextgroup=xIfCondition skipwhite
:syntax match xIfCondition /([^)]*)/ contained nextgroup=xThen skipwhite
:syntax match xThen /then/ contained
```

"nextgroup"参数指定哪些组可以跟在该组后面. 这个参数并不是必需的. 如果由它指定的任何一个组都不符合匹配. Vim什么也不做. 比如下面的代码：

code
```
        if not (condition) then
```

"if"是匹配到了xIf组. 但"not"却不符合由nextgroup指定的组xIfCondition，所以这样一来只有"if"被正确高亮.

"skipwhite"参数告诉Vim空白字符(空格和跳格键)可以出现在语法项之间. 另一个与类类似的参数是"skipnl", 它允许在语法项之间出现断行, "skipempty"参数, 允许出现空行, 注意"skipnl"并不会跳过任何空行, 它要求断行之后必需有东西被匹配到才行.

44.7 其它参数

匹配一个区域

一旦定义一个区域, 整个区域就会应用由组名指定的高亮规则. 比如, 要高亮括号()里面的名为xInside的语法项, 使用下面的命令:

ex command
```
:syntax region xInside start=/(/ end=/)/
```

如果你想以不同的规则来高亮括号. 当然可以写一个复杂的区域定义语句来完成它, 但"matchgroup" 参数带来一种更简单的办法. 它告诉Vim以另外一种高亮组来处理区域的首尾部分.(本例中, 用xParen组):

ex command
```
:syntax region xInside matchgroup=xParen start=/(/ end=/)/
```

"matchgroup"参数应用于其后的start或end模式. 上例中区域的首尾都以xParen组来高亮. 也可以分别为它们指定不同的组:

ex command
```
:syntax region xInside matchgroup=xParen start=/(/
        \ matchgroup=xParenEnd end=/)/
```

"matchgroup"的副作用是位于区域首尾的嵌套的语法项不能被正确识别了. "transparent"相关的例子涉及到了这个问题.

透明语法

你可能希望C程序中"while"之后的()和"for"之后的()以不同的颜色来显示. 这两种类型的循环语句中的()中都可以包含嵌套的()语法项, 这些语法项也应该以同样的方法进行高亮. 所以必需确保配对的()进行适当的高亮. 下面是一种方案:

ex command
```
:syntax region cWhile matchgroup=cWhile start=/whiles*(/ end=/)/
        \ contains=cCondNest
:syntax region cFor matchgroup=cFor start=/for\s*(/ end=/)/
        \ contains=cCondNest
:syntax region cCondNest start=/(/ end=/)/ contained transparent
```

现在你可以让cWhile和cFor拥有不同的语法高亮了. cCondNest可以嵌套出现在其中，但是它所应用的高亮规则将是包含它的语法项，这是"transparent"的作用.

注意例子中的"matchgroup"的名字与定义的语法项名字相同. 为何这样定义?这样使用matchgroup的一个副作用是被包含的语法项不能是第一个语法项. 这样避免了cCondNest组匹配到"while"或"for"之后的第一个(，否则的话，cCondNest就会一直匹配到与第一个"(对应的")". 现在cCondNest 只可能匹配第一个"("之后的语法项.

偏移

如果你想定义一个区域匹配"if"之后的"()"之间的内容. 但是却不包含"if"和"(", ")"本身. 这可以通过指定匹配模式中的偏移来实现. 例如:

ex command

```
:syntax region xCond start=/ifs*(/ms=e+1 end=/)/me=s-1
```

起始模式的偏移是"ms=e+1". "ms"代表"Match Start". 它定义了一个自目标字符串起始位置的偏移. 通常匹配的起始位置就是模式目标的起始位置. "e+1"则告诉Vim匹配的起始位置始自模式目标的结束处再向前偏移一个字符[1]

结束模式的偏移是"me=s-1". "me"指"Match End". "s-1"意为模式目标的起始处的上一个字符. 结果是对于程序语句[2]只有"foo == bar"会被应用xCond语法高亮.

关于偏移的更多详情请参考: |:syn-pattern-offset|.

单行

"online"参数告诉Vim要匹配的区域不能跨越多行. 比如:

ex command

```
:syntax region xIfThen start=/if/ end=/then/ oneline
```

定义的是这样一个区域，它以"if"开始，以"then"结束. 但是如果同一行上"if"之后没有"then"，那就不会匹配到.

[1]译: 对于if(foo == bar) 这样的语法，上面的例子就不够用了. 可以改为:
:syntax region xCond start=/if\s*(s*/ms=e+1 end=/\s*)/me=s-1
毕竟空白字符不应该应用高亮颜色

[2]译: 上下前后，似乎都会有歧义，文档中规定"1234"中，2是1的下一个字符/后一个字符，1是2的上一个字符/前一个字符.

备注：在区域定义中用了"oneline"之后如果同一行中没有找到结束模式．那匹配就不会发生．如果没有"oneline"关键字Vim就不会检查结束模式是否匹配．所以即使没有找到其结束模式，区域匹配也会成功．

后续行

现在问题稍复杂一点．我们来定义一个预处理语法．预处理行以第一列的#开始，直到该行结束．同时以\结束的行又指示到下一行将延续未竟的预处理定义．要应付这种情况就需要在语法项中包含一个指示后续行的模式：

ex command
```
:syntax region xPreProc start=/^#/ end=/$/ contains=xLineContinue
:syntax match xLineContinue "\\$" contained
```

上例中，虽然xPreProc匹配一个单行，但它所包含的匹配组(即xLineContinue)却使其可以继续匹配多行，比如，匹配下面的行：

code
```
#define SPAM  spam spam spam \
                bacon and spam
```

这正是我们要的效果．如果还有差强，你可以定义一个区域，通过在它所包含的子模式中指定"excludenl"关键字来限制只在行尾匹配一个模式．比如，要在xPreProc中高亮"end"，但只是针对于行尾．为避免xPreProc象xLineContinue一样在下一行延续，可以这样使用"excludenl"：

ex command
```
:syntax region xPreProc start=/^#/ end=/$/
        \ contains=xLineContinue,xPreProcEnd
:syntax match xPreProcEnd excludenl /end$/ contained
:syntax match xLineContinue "\\$" contained
```

"excludenl"必需出现在匹配模式之前．这样"xLineContinue"才不致受"excludenl"的影响，它还可以象以前一样把xPreProc延续到后续行去．

44.8 簇

开始写大批Vim语法文件时你需要记住一件事．Vim中可以定义一些语法组为一个簇．假设你有一种语言包含了诸如循环，if语句，while循环和函数定义这样的语法项．其中每个又包含了一些共同的语法元素：数字和标识符．你可以这样来定义：

ex command

```
:syntax match xFor /^for.*/ contains=xNumber,xIdent
:syntax match xIf /^if.*/ contains=xNumber,xIdent
:syntax match xWhile /^while.*/ contains=xNumber,xIdent
```

每次定义一个语法项都需不胜繁琐地用"contains="来罗列它所包含的语法元素. 如果你想增加另一个语法元素, 又得在3个地方都加上. 语法簇可以以一个簇名代替这些众多的语法元素. 下面的例子定义了一个包含了两个语法元素的语法簇:

ex command

```
:syntax cluster xState contains=xNumber,xIdent
```

簇被用在其它以syntax 语法定义语句中. 就象语法组一样. 它们的名字以@开始. 这样, 你可以这样简化上面的例子:

ex command

```
:syntax match xFor /^for.*/ contains=@xState
:syntax match xIf /^if.*/ contains=@xState
:syntax match xWhile /^while.*/ contains=@xState
```

下面语句使用"add"把新的语法元素加到一个簇中:

ex command

```
:syntax cluster xState add=xString
```

类似地, 下面是从簇中移除一个语法项:

ex command

```
:syntax cluster xState remove=xNumber
```

44.9 包含另一个语法文件

C++的语法是C语言的一个超集. 毕竟人们都不希望把相同的东西写上两遍, 所以Vim提供了在一个语法中读取另一个语法文件的功能:

ex command

```
:runtime! syntax/c.vim
```

":runtime!"会在由'runtimepath'指定的目录中寻找"syntax/c.vim"文件. 这样可以借用所有为C文件定义的C语言语法. 如果你替换了c.vim语法文件或以另一个文件来补充它, 它们也会如影随形地对C++发生同样的影响.

载入了所有的C语言项后就可以这样C++的特有部分了, 比如增加C里面没有的关键字:

ex command
```
:syntax keyword cppStatement    new delete this friend using
```

现在我们来关注一下Perl语言. Perl语言允许同时存在两个部分: 一个POD格式的文档节, 一个是程序本身. 其中POD节以"=head"开始, 以"=cut"结束[1]

你可以在一个文件中定义POD的语法, 然后从Perl本身的语法文件中引用它. ":syntax include"命令从另一个文件中读取语法定义, 并把其中定义的语法元素组织为一个簇保存起来. 如:

ex command
```
:syntax include @Pod <sfile>:p:h/pod.vim
:syntax region perlPOD start=/^=head/ end=/^=cut/ contains=@Pod
```

一旦在Perl文件里发现了"=head", perlPOD语法项所定义的区域开始. 该语法项的定义包含了一个名为@Pod的簇. 所有在pod.vim中定义的顶级元素都是该簇的一部分. 找到"=cut"后, 区域匹配结束, 继续处理Perl中定义的其它语法项.

":syntax include"命令会聪明地忽略掉被包含文件中的":syntax clear". 同时象"contain=ALL"这样的参数也只会包含在当前文件中的语法项, 而不会包含在include当前文件的父文件中所定义的语法项.

"<sfile>:p:h/"中用到了代表当前文件的(<sfile>), 并且扩展到该文件的绝对路径(:p)然后又取该绝对路径的目录部分(:h). 结果是将同一个目录下的pod.vim文件包含进来.

44.10 语法同步

编译器处理起语言的语法易如反掌, 毕竟这是它的本职工作. 只需将文件从头至尾进行解析. Vim却要为此犯难. 它必需能断章取义, 程序写到哪里, 它就要跟到哪里. 它又是怎么确定何处要回承转合, 何处要适可而止呢?

答案是":syntax sync"命令. 它告诉Vim如何确定当前的境况. 比如, 下面的命令告诉Vim回溯查找C风格的注释, 并从注释的起始处开始对该语法项进行着色:

ex command
```
:syntax sync ccomment
```

此外还可以以一些参数调整该命令的处理细节. "minlines"参数告诉Vim最少要往回查找多少行, "maxlines"告诉编辑器最多检查多少行.

[1]译: POD: Plain Old Documentation

比如，下面的命令告诉Vim至少要往回看10行：

ex command
```
:syntax sync ccomment minlines=10 maxlines=500
```

如果往回查看还未能找到期望的语法模式，那就继续回溯直到找到目标模式． 但是它最多也不会回溯超过500行． （大的"maxlines"会让Vim处理语法高亮速度减慢． 值太小又可能导致着色有误）

为了使Vim同步着色更快一些，可以让它跳过一些语法项．那些实际要显示文本时才用到匹配可以加上"display"参数．

默认情况下，注释项会以Comment语法组的颜色设定进行着色．如果你想使用另外的颜色方案，可以为它指定一个不同的语法组：

ex command
```
:syntax sync ccomment xAltComment
```

如果你要定义的编程语言中的注释还是C风格的，可以用另一种方法进行显色同步．最简单的莫过于告诉Vim回溯一些行然后从那里开始解析．如下面的命令告诉Vim 从前150行处开始进行解析：

ex command
```
:syntax sync minlines=150
```

"minlines"设置太大会让Vim变慢，尤其是你在浏览文件过程中需要频频往回滚动时．

最后，你还可以指定在回溯解析时要定位的目标语法组，如：

ex command
```
:syntax sync match {sync-group-name}
        \ grouphere {group-name} {pattern}
```

这告诉Vim以匹配{pattern}的位置为语法组的开始． {sync-group-name}用来为该项语法同步指定一个名字． 例如，下面的shell脚本中定义了一个if语句，以"if"开始，以"fi"结束：

code
```
if [ -f file.txt ] ; then
        echo "File exists"
fi
```

下面的命令为该语法指定了"grouphere"指示符：

ex command
```
:syntax sync match shIfSync grouphere shIf "<if>"
```

"groupthere"参数则告诉Vim什么情况下一个语法组结束. 如, if/fi组的结束可以这样定义:

ex command
```
:syntax sync match shIfSync groupthere NONE "<fi>"
```

此例中的NONE告诉Vim当前位置不属于任何的语法区域. 特别是不属于if块

同样也可以定义没有"grouphere"和"groupthere"的模式和区域. 这些组将在语法同步时被简单地跳过去. 例如, 下面的定义跳过了任何{}中的内容, 即使它可能会符合其它的语法同步定义:

ex command
```
:syntax sync match xSpecial /{.*}/
```

关于语法同步的更多内容在参考手册中, 请参阅: |syn-sync|.

44.11 安装一个语法文件

当你有一个新的语法文件时, 把它放在'runtimepath'指定的路径下名为"syntax"的目录.在Unix系统上典型的是"~/.vim/syntax".

语法文件的名字必需与文件类型名一致, 以".vim"为扩展名. 对于x语言, 其完整的路径应该是:

List
```
~/.vim/syntax/x.vim
```

同时你必需让Vim能正确识别该文件类型. 请参考|43.2|.

如果你的语法文件用起来感觉不错, 你可能还想造福其它的Vim用户. 请先读完下一小节确保你的文件对别人也能正常工作. 然后把它email给Vim的当前维护者: maintainer@vim.org. 同时在信中解释一下该文件类型是如何被检测的. 幸运的话你写的文件也可能出现在Vim的下一个版本中!

向已有的语法文件中添加内容

上面我们都是假设你加的是一个全新的文件. 如果已有一个语法文件可用, 只是稍有欠缺时, 你可以在另外一个文件中添加一些定义. 这样可以避免修改那些已经发布的语法文件, 因为这些文件在安装一个新版Vim时便会被覆盖掉.

你可以在文件中直接使用语法命令, 很可能还要引用已有语法文件中的语法组. 比如, 下面的例子向C语法文件添加新的变量类型:

ex command
```
:syntax keyword cType off_t uint
```

新添加的语法文件要与原来的语法文件同名． 上例中就应该是"c.vim"．然后把它放在'runtimepath'指定的靠后的某个目录． 这样可以保证它的载入顺序是在原语法文件之后． 对Unix系统可以是：

List
```
~/.vim/after/syntax/c.vim
```

44.12 可移植的语法文件

如果所有的Vim用户都可以交互他们的语法文件该有多好？要做到这一点，语法文件就必需遵循下面的规则．

首先在文件头加一段注释说明该文件的用途，它的维护人以及最后更新时间． 不要在这里放入太多的修订列表，没人会看这些． 下面是一个样板：

Display
```
" Vim syntax file
" Language:     C
" Maintainer:   Bram Moolenaar <Bram@vim.org>
" Last Change:  2001 Jun 18
" Remark:       Included by the C++ syntax.
```

写其它语法文件时也请使用上面的模板．利用一个已经存在的文件会节省大量时间．

为你的语法文件选一些描述性强的好名字． 在名字中可以用小写字母和数字． 不要太长，这个名字会在多处使用： 语法文件名"name.vim"，'filetype'，b:current_syntax以及每个语法组的开始(如nameType，nameString，nameString等等)．

记得在文件里检查"b:current_syntax"． 如果该变量已经定义，那说明'runtimepath' 里更靠前的某个目录里已经载入过了该语法文件．要与Vim5.8版本兼容，可以用下面的样例： 在文件尾可以设置"b:current_syntax"变量为当前文件所定义的语法的名字． 别忘了在被包含的文件里也会设置该变量，如果你包含了两个文件那就要重置"b:current_syntax"．

如果你想让你的语法文件兼容于Vim5.x版本，那就要另外检查v:version变量． 请参考yacc.vim．

不要包含属于用户个人偏好的设置．不要设置'tabstop'，'expandtab'等等诸如此类的选项．它们应该出现在文件类型plugin里．

也不要设置映射和缩写. 必要的话可以设置'iskeyword'来识别关键字.

尽量不要用特殊的颜色. 最好是链接到标准的语法高亮组. 别忘了有些人喜欢用另类的背景色, 或者他只能用8种颜色. 为了与Vim 5.8版兼容可以使用下面的样例:

code
```
if version >= 508 || !exists("did_c_syn_inits")
  if version < 508
    let did_c_syn_inits = 1
    command -nargs=+ HiLink hi link <args>
  else
    command -nargs=+ HiLink hi def link <args>
  endif

  HiLink nameString     String
  HiLink nameNumber     Number
  ... etc ...

  delcommand HiLink
endif
```

记着为那些不需要语法同步的语法项加上"display"参数, 这样可以加快往回滚动和按下CTRL-L时的处理速度.

下一章: |usr_45.txt| 选择语言
版 权: 请参考|manual-copyright| vim:tw=78:ts=4:ft=help:norl:

usr_45.txt Vim 6.3版.最后修改: 2002年10月08日

VIM用户手册--- 作者: Bram Moolenaar

选择语言

Vim中的消息可以以多种语言显示. 本章讲述如何切换显示的语言以及处理不同语言的方法.

下一章: |usr_90.txt| Vim安装
前一章: |usr_44.txt| 自定义语法高亮
目 录: |usr_toc.txt|

45.1 用于消息的语言

Vim启动时, 它会去检查你的系统所用的语言. 多数情况下都没问题, 它将以你的本土语言显示消息(如果相应语言已安装好的话). 下面的命令可以查看当前所用的语言:

ex command
```
:language
```

如果结果是"C", 那就是说用的是默认语言, 英语.

> 备注: 只有Vim编译时打开了支持多语言显示时才能使用该功能. 要查看当前的Vim是否支持该功能, 可以检查":version"命令看有没有"+gettext"和"+multi_lang". 如果有就没事, 如果看到的是"-gettext"或"-multi_lang"那就坏菜了, 你得再找一个Vim试试.

如何显示几种语言的消息呢? 好几处方法, 具体要看你用的是什么系统.

第一个办法是启动Vim之前把你的环境设为想用的语言. 比如在Unix上可以这样:

shell command
```
env LANG=de_DE.ISO_8859-1  vim
```

这也只有你的系统上已经支持该语言时才能奏效. 好处是接下来GUI消息和函数库都能使用这个正确的设置了. 缺点是必需在Vim启动之前设置好. 如果你想在Vim运行时改变设置,那可以用第二种方法:

ex command
```
:language fr_FR.ISO_8859-1
```

用这种办法你可以用好几个名字来代表你要用的语言. 系统不支持该语言的话会有一个错误消息. 如果只是系统中没有对应该语言的翻译文本. Vim会一声不响地返回到默认的英语.

要查看系统中就支持哪些语言,可以先找到存放语言设置的目录. 在我的系统上是"/usr/share/locale". 在一些系统上位于"/usr/lib/locale". "setlocale"的帮助文档应该会告诉你它在当前系统上的位置.

注意要一字不差地输入对应的名字. 大小写,"-"和"_"都不能弄错.

你可以把系统中的消息文本,编辑文本和时间格式分别设置为不同的语言. 请参考|:language|.

DIY--自己翻译

如果你选的语言中没有对应的消息文本. 你可以自己来写. 首先要拿到Vim的源代码和GNU的gettext软件包. 把源代码解压后,可以在src/po/README.txt里找到相应的操作指南.

自己翻译没想象中那么难. 你不一定非得懂编程才行. 不过当然你得懂英语和你要翻译的目标语言.

如果你对自己的翻译成果感觉良好,可以考虑让大家共享. 把它上载到vim的web站点(http://vim.sf.net)或email给Vim的维护人<maintainer@vim.org>. 或者都给他们来一份!

45.2 用于菜单的语言

默认的菜单是英语. 要用你的本土语言显示菜单,必需先进行翻译. 如果环境变量已经设置好的话通常这些都会自动弄好,就象消息文本一样. 不需要额外再做什么. 但是这一切都必需是在相应的内容都已翻译好的前提下.

假设你在德国,语言设置为德语,但是又希望菜单显示"File"而不是"Datei". 你可以用下面的命令换回英语菜单:

ex command
```
:set langmenu=none
```

当然也可以选用另一种语言:

ex command
```
:set langmenu=nl_NL.ISO_8859-1
```

正如上面的命令中一样, "-"和"_"不能弄混. 不过此处的大小写倒是无关紧要.

'langmenu'选项必需在载入菜单之前设置. 一旦菜单已经载入再去'langmenu'选项就不会直接看到效果. 所以, 最好把'langmenu' 放在你的vimrc文件里.

如果你坚持要在Vim运行时切换菜单语言, 可以这样做:

ex command
```
:source $VIMRUNTIME/delmenu.vim
:set langmenu=de_DE.ISO_8859-1
:source $VIMRUNTIME/menu.vim
```

这样有一个损失: 所有你自己定义的菜单都会丢掉. 你要重新定义.

自己翻译菜单

要查看哪些语言已经有了对应的菜单翻译, 请检查目录:

List
```
$VIMRUNTIME/lang
```

文件名为menu_{language}.vim. 如果你没看到有对应语言的文件名, 那你又有机会自己翻译了. 最简单的办法是把某个文件复制过来进行修改.

首先用":language"命令找到你的语言. 在文件名{language}中就用这个名字, 不过记住要小写. 然后把它复制到你自己的运行时目录, 可以通过前面介绍过的'runtimepath'找到. 比如, 在Unix上你可以这样:

ex command
```
:!cp $VIMRUNTIME/lang/menu_ko_kr.euckr.vim ~/.vim/lang/menu_nl_be.iso_8859-1.vim
```

你也可以在"$VIMRUNTIME/lang/README.txt"文件里找到行动指南.

45.3 使用另一种编码方法

Vim会猜测你正在编辑的文件会用当前语言相应的编码方案. 对多数欧洲语言来说是"latin1". 每个字节一个字符. 也就是说一共有256种不同的字符. 对亚洲语种来说这可远远不够. 它们通常用的是双字节的编码方案, 可以提供上万种不同的字符. 即使这样, 同一个文件包含多种语言时还是不够用. 这就是为什么要用Unicode的原因. 它被设计来容纳常用语言的所有字符. 这是一种替代所有其它方案的超级编码. 但是目前还未成大器.

幸运的是, Vim支持上面的3种编码方案. 除了有一些限制外, 你甚至可以在环境变量设置为另一语言时使用它们.

而且, 如果你只是编辑你的本土语言的话, 默认的编码方案应该就没错, 你什么都不用做. 下面的内容只是针对你想以不同语言编辑文件的情况.

> 备注: 只有Vim编译时进行相应设置才可使用多种编码方案. 要看是否支持该功能, 可以检查":version"的输出中有没有"+multi_byte". 如果有就没事. 如果是"-multi_byte"你就要另找一个Vim了.

中GUI中使用UNICODE

使用Unicode的好处是其它编码方案可以和它相互转换, 又不会丢失信息. 如果你的Vim 内部本身就是用Unicode, 你就可以使用任一种编码方案.

不幸的是, 支持Unicode的系统毕竟有限. 所以你所选的语言可能就没用它. 这时就需要你明确告诉Vim你想用Unicode, 以及如何处理系统中其它的界面部分.

我们来启动GUI版的Vim, 它可以显示Unicode字符. 下面的命令应该可以工作:

ex command

```
:set encoding=utf-8
:set guifont=-misc-fixed-medium-r-normal--18-120-100-100-c-90-iso10646-1
```

'encoding'选项告诉Vim你所用的字符是什么. 它同时影响缓冲区内的文件(你编辑的文件) , 寄存器的内容以及Vim的脚本文件等. 你可以把'encoding'看成对Vim的内部设置.

上例假设你的系统上已经安装好了相应的字体. 例子中的名字是X-Windows上的. 该字体位于一个增强xterm Unicode功能的包中. 如果你没有该字体, 可以从下面下载:

URL
```
http://www.cl.cam.ac.uk/~mgk25/download/ucs-fonts.tar.gz
```

对MS-Windows来说，一些字体中Unicode字符有限．试一下"Courier New"字体．你可以使用"Edit/Select Font..."菜单命令来选择字体．注意只能用等宽字体．如：

ex command
```
:set guifont=courier_new:h12
```

如果不能正确显示，取得fontpack试试．如果微软的链接还没改动，你可以在此处下载：

URL
```
http://www.microsoft.com/typography/fontpack/default.htm
```

现在你已经告诉Vim以Unicode作为内部编码，并以Unicode字体显示文本． 输入的字符仍然根据原来的语言设置进行编码． 这需要把它们转换为Unicode编码． 'termencoding' 选项可以告诉Vim从何种语言进行转码． 比如下面的命令：

ex command
```
:let &termencoding = &encoding
:set encoding=utf-8
```

上面的命令把'encoding'选项的值赋予'termencoding'选项．然后将'encoding'设置为utf-8． 你需要在自己的版本上试一试．如果你是用亚洲语系而又希望编辑Unicode文本． 这一功能就显得尤其有用．

在支持Unicode的终端上使用Unicode

有一些终端可以直接支持Unicode． 随XFree86发布的标准xterm就是一例． 下面我们就以此为例． 首先，xterm必须在编译时打开了Unicode支持功能．参考|UTF8-xterm|看如何进行编译．以"-u8"参数启动xterm． 最好同时指定字体．如：

shell command
```
xterm -u8 -fn -misc-fixed-medium-r-normal--18-120-100-100-c-90-iso10646-1
```

现在你可以在该终端中直接支持Vim． 象前述的命令一样把'encoding'设为"utf-8"． 就是这么简单．

在普通终端中使用Unicode

如果你要处理Unicode文本，而当前的终端并不支持Unicode． 你还可以在Vim中使用，虽然终端不能支持的字符将不会正常显示． 但整个文本的布局还是会保持良好．

ex command
```
:let &termencoding = &encoding
:set encoding=utf-8
```

看起来这与GUI中的情况一样，实际上两者作了不同的处理： Vim会在把字符送至终端之前进行转换．以免弄乱了文本的外观．

这样的转换要想行得通还必需保证'termencoding'到'encoding'的转换是可行的． Vim 可以把latin1转换为Unicode，这样的转换总是没有问题．进行其它的转换则必需保证|+iconv|这一特性．

试一下在一个文件中以Unicode字符进行编码． 你会发现Vim以一个问号来替代无法显示的字符(或者是下划线或其它字符)．将光标移至该字符上用下面的命令试试：

normal mode command
```
ga
```

Vim会显示一行信息，报告该字符的身家底细． 它会告诉你当前这个字符到底是什么． 你也可以在一个Unicode表里看到它． 实际上你有时间的话可以慢慢浏览该文件．

> 备注： 因为'encoding'用于Vim中所有的文本，所以改变该选项可能会引起非ASCII字符无效． 使用寄存器和'viminfo'的时候你就会体会到这一点(比如，一个记录下来的搜索模式)． 所以推荐你在vimrc配置文本里设置好'encoding'后就别再动了．

45.4 编辑另类编码方案的文件

如果你在安装Vim时指定它使用Unicode，然后你希望编码一个16-bit的Unicode文本． 听起来顺理成章，果真如此简单？ 实际上，Vim在内部总是使用utf-8编码，所以16-bit的编码必需进行转换．字符集(Unicode)和编码方案(utf-8或utf-16)之间还是有一些差别的．

Vim会自动检测你的文本． 它将采用'fileencodings'选项中指定的可用编码方案．使用Unicode时，其默认值是"ucs-bom,utf-8,latin1"．这意味着Vim 会检查文件看它使用了这三种编码方案的哪一种．

List

ucs-bom	文本必需以一个字节顺序的标记进行标识．这使得检测16-bit还是32-bit以及utf-8成为可能．
utf-8	utf-8编码． 如果发现某个字符序列中出现了非utf-8字符就不能选用该方案．
latin1	老式的8-bit编码． 仍然运作良好．

如果你在编辑一个16位的Unicode 文件，其中含有一个字节顺序标记，Vim会检测到该文件的编码类型并在读取时将其转换为utf-8． 'fileencoding'选项(注意没有s)被设置为检测到的编码． 在此例中就是"ucs-2le"． 这是说该文件是Unicode编码，每个字符占两个字节，使用的是little-endian．这种文件格式在MS-Windows系统上很常用(比如注册表文件)．

保存文件时，Vim会比较'fileencoding'和'encoding'． 如果两者不同，文件内容就会先被转换．

如果'fileencoding'选项的值为空就那是说无需进行转换．即假设文件是以'encoding' 编码的．

如果默认的'fileencodings'设置对你并不合适，你可以把它定义为你希望Vim识别的编码集． 只有在发现当前编码不符时Vim才会试用下一个．把"latin1"放在首位就不行，因为它永远都不会是非法的． 下例中，文件没有字节顺序标记并且不是utf-8时设为日文：

ex command
```
:set fileencodings=ucs-bom,utf-8,sjis
```

请参考|encoding-values|了解Vim建议的设置．另外的设置也有可能是可行的，这要看是否能进行编码转换而定．

强制编码

如果自动编码识别失败，你就必需告诉Vim文件的编码方案．如：

ex command
```
:edit ++enc=koi8-r russian.txt
```

"++enc"部分使Vim仅将该文件视为指定编码方案．Vim会将指定的编码方案，此例中是俄语，转换为'encoding'编码方案，同时'fileencoding'选项也被设为指定的编码方案，以便保存文件时进行逆向转换．

保存文件时同样可以使用该参数． 通过这种方法你也可以让Vim来进行文件格式转换．如：

ex command
```
:write ++enc=utf-8 russian.txt
```

备注： 文件格式转换可能会引起内容丢失．从其它编码转换为Unicode一般来说没有什么问题，除非其中有非法字符．而从Unicode转换到其它编码格式则经常会引起信息丢失，尤其是被转换的文件中使用了一种以上的字符集时．

45.5 输入

计算机的键盘按键不过百余. 但一些语言则有上千的字符, Unicode更是有1万的字符. 这么多的字符是如何输入的呢?

首先, 如果文件中要用的特殊字符不多的话, 可以使用digraphs. 这在|24.9|中已经解释过了.

如果你所用的语言对应的字符多于键盘的键数, 你就需要借助于某种输入法. 当然你得学习这种输入法. 需用输入法的话你可能已经安装好了. 它应该能在Vim中正常工作, 就象它支持其它程序一样. 更多详情请参考|mbyte-XIM|(适用于X-Windows), 和|mbyte-IME|(适用于MS-Windows).

键映射[1]

有些语言用的字符集不是latin, 但字符个数都差不多. 这时可以通过映射键进行处理. Vim使用键盘映射的办法.

假设你想输入希伯莱文字. 你可以载入它对应的键盘映射:

ex command
```
:set keymap=hebrew
```

Vim会试着找到一个键映射文件. 这视'encoding'设置而定. 如果没有找到符合的映射文件, 则会报告一个错误.

现在你可以在Insert模式下输入希伯莱文字了. 在Normal模式和冒号命令行模式, Vim 会自动转换为常规的英语键盘布局. 你可以用下面的命令在两种模式间切换:

normal mode command
```
CTRL-^
```

这一命令只对Insert模式和Command-line格式有效. 在Normal模式下它的意义完全不同(跳转到最近编辑的文件).

如果'showmode'选项打开的话, 键映射的用法会显示在模式信息中. 在GUI版的Vim中还会以不同的颜色显示.

你也可以用'iminsert'和'imsearch'两个选项来改变键映射的用法.

下面的命令可以查看映射的列表:

ex command
```
:lmap
```

[1]译: 此处的键映射与作为命令快捷键的键映射是两个不同的概念

要找出就有哪些映射可用，在GUI版本下可用Edit/Keymap菜单． 其它情况可以用下面的命令：

ex command
```
:echo globpath(&rtp, "keymap/*.vim")
```

键映射DIY

你可以创建自己的键映射文件．这并不很难．先找一个与你的目标比较接近的语言． 把它复制到Vim运行时目录下的"keymap"子目录． 比如，在Unix系统上，你可以放在"~/.vim/keymap"．

键映射文件的名字必需是形如：

List
```
        keymap/{name}.vim
or
        keymap/{name}_{encoding}.vim
```

的格式其中的{name}是键映射的名字． 选一个好名字吧，但记住不要与已有的重名(除非你想把它替换掉)．名字里面不能含有下划线．此外还可以跟一个可选的编码格式，如：

List
```
        keymap/hebrew.vim
        keymap/hebrew_utf-8.vim
```

定义后的文件应该是自说明的． 可以参考一下随Vim发行的那些键映射文件．详情请参考|mbyte-keymap|．

最后一招

如果上面的所有办法都失灵，你还可以用CTRL-V输入任何字符：

List

encoding	type	range
8-bit	CTRL-V 123	decimal 0-255
8-bit	CTRL-V x a1	hexadecimal 00-ff
16-bit	CTRL-V u 013b	hexadecimal 0000-ffff
31-bit	CTRL-V U 001303a4	hexadecimal 00000000-7fffffff

例子中的空格不能键入． 详情请参考|i_CTRL-V_digit|．

下一章： |usr_90.txt| Vim安装

usr_90.txt Vim 6.3版.最后修改: 2004年03月21日

VIM用户手册--- 作者: Bram Moolenaar

Vim安装

install

能用上Vim之前你必需先安装它. 安装的简易程度依你的操作系统而定. 本章为你的安装提供一些向导，同时教你如何升级到一个新的版本.

前一章: |usr_45.txt| 选择语言
目 录: |usr_toc.txt|

90.1 Unix

首先你要搞清楚是为整个系统安装Vim还是为单个用户安装. 安装过程倒是几乎一样，不过安装后的目录对这两种情况来说有所不同.

为整个系统安装Vim的话通常用的基准目录是"/usr/local". 但也可能因你所用的系统不同而异. 你可以试着看看其它软件包都安装在哪个目录.

为单用户安装Vim时，你可以把该用户的home目录作为基准目录. 安装文件将会被放在其下名为"bin"和"shared/vim"的目录中.

从一个预编译包中安装

多种UNIX系统都有预编译好的包可供安装. 参考下面URL中的列表.

http://www.vim.org/binaries.html

这些二进制包由志愿者们维护，有可能已经过期了. 直接从源代码编译你自己的UNIX 版本的Vim是个不错的选择. 同时，直接从源代码打造你的Vim也可以让你控制各种特性的取舍. 当然，说这些东西之前编译器是必备的.

如果你用的是Linux，"vi"程序一般来说指向一个简装版的Vim. 它没有语法高亮功能. 你可以试着在你的Linux光盘里找到另一个Vim软件包，或者从网上搜索.

从源码构造

要编译安装Vim，你需要有以下准备:

```
- 一个C编译器 (最好是GCC)
- GZIP程序(你可以从www.gnu.org下载)
- Vim的源代码和运行时文件
```

要得到Vim的文档，请查看下面给出的镜像站点，你应该能从中找到对你最快的站点:

ftp://ftp.vim.org/pub/vim/MIRRORS

或者访问从主站点ftp.vim.org，如果你觉得它已经够快的话. 在"unix"目录下你会看到很多文件. 版本号跟文件名连在一起. 往往最想下载的总是最新版.

一共有两种办法得到这些文件: 一个包括了所有东西的大的文档，或者4个分开的文档，每个可以放在一张磁盘上. 对6.1版本而言那个单个的大文档是:

```
vim-6.1.tar.bz2
```

这需要bzip2程序来解压. 如果你没有这个程序的话，去抓取那4个小一点的文件，它们是用gzip压缩的. 对于Vim 6.1版来说这4个文件名是:

```
vim-6.1-src1.tar.gz
vim-6.1-src2.tar.gz
vim-6.1-rt1.tar.gz
vim-6.1-rt2.tar.gz
```

编译

首先创建一个顶层的工作目录，比如:

```
mkdir ~/vim
cd ~/vim
```

然后把压缩文件解到该目录下． 如果是单个的大文档，可以用

shell command
```
bzip2 -d -c path/vim-6.1.tar.bz2 | tar xf -
```

来解压．然后切换路径到你下载这些文件的目录下．

shell command
```
gzip -d path/vim-6.1-src1.tar.gz | tar xf -
gzip -d path/vim-6.1-src2.tar.gz | tar xf -
gzip -d path/vim-6.1-rt1.tar.gz | tar xf -
gzip -d path/vim-6.1-rt2.tar.gz | tar xf -
```

如果你对默认的特性已经心满意足，系统环境也没问题的话，你应该可以直接以这样的命令开始编译:

shell command
```
cd vim61/src
make
```

make程序会运行配置程序并且进行所有必要的编译．等下我们会介绍如何把各种特性编译进去．

如果编译过程发生了错误，请仔细检查给出的错误信息．这些信息往往指明了错误的原因．但愿你能根据这些自己解决问题．可能要关闭一些功能来使编译通过．查看Makefile中对你所用的特定系统的提示．

测试

现在你可以用命令

shell command
```
make test
```

检查一下编译后的程序是否正常． 这个命令将会运行一系列的测试脚本来检查是否与预期的结果相符合．在此过程中Vim会多次启动，屏幕上会有很多文本信息一闪而过．如果一切顺利的话你会最终看到这样的字样:

Display
```
test results:
ALL DONE
```

如果有一两个错误的话，Vim还是可以工作，只是不够完美而已． 如果你看到了太多的错误信息或者Vim停不下来，那可能是有地方错了． 要

么你自己能解决它，或者你可以找人来帮忙．你可以检查|maillist-archive|看有没有现成的解决方案．如果还有其它东西失败了．你可以在|maillist|里提问，看看有没有人能帮你．

安装

install-home 如果你想安装在你的home目录，编辑Makefile文件，查找这样的行：

Display
```
#prefix = $(HOME)
```

移去最前面的#．

如果是为整个系统安装，Vim极有可能已经自己检测到了一个合适的安装目录．你也可以自行指定一个，需要有root权限．

要安装Vim执行：

shell command
```
make install
```

这应该会让相关的文件都安装就绪．现在可以试着运行一次来验证一下是否装好了．下面两个简单的测试可以证明运行时文件是否也装好了：

ex command
```
:help
:syntax enable
```

如果不行，这个命令可以告诉你Vim是在哪找运行时文件的：

ex command
```
:echo $VIMRUNTIME
```

下面的命令也可以让你得知启动过程中发生的事情：

shell command
```
vim -V
```

别忘了用户手册都假设你以某种方式使用Vim．装好后，遵照|not-compatible|中的指示来让你的Vim能象手册说的那样．

选择哪些功能

有多种办法来取舍某个Vim功能，最简单的莫过于直接编辑Makefile文件．该文件里本身就以注释的形式嵌入了很多关于如何编辑它的建议和例子．通常你可以通过为一行文本加上注释或去掉它的注释来决定某个功能的去留．

另一个办法是运行"configure"程序. 这可以让你手工指定配置选项. 不过你得一字不差地键入整个命令.

下面是一些最有用的配置选项. 它们也可以通过修改Makefile里的相关内容来实现.

List
```
--prefix=directory        安装Vim的顶层目录

--with-features=tiny       关闭多数功能进行编译
--with-features=small      关闭部分功能
--with-features=big        打开多数功能
--with-features=huge       打开几乎所有功能
                           参考 |+feature-list| 可以了解更多关于
                           一些更多明细的功能特性.

--enable-perlinterp        启用Perl接口. 类似的还有ruby, python
                           和tcl的接口.

--disable-gui              不把GUI编译进去
--without-x                不编译X-windows支持
                           这两项都被关闭的话, Vim就不会连到一个X
                           服务器上了, 这会让启动快一些.
```

别忘了用帮助:

shell command
```
./configure --help
```

它可以告诉你全部可用的选项. 你还可以在|feature-list|找到每个功能对应的简单解释以及其它相关链接和信息.

乐于冒险的话, 你也可以自己编译头文件"feature.h". 你甚至可以自己修改源码!

90.2 MS-Windows

有两种办法可以安装Vim的MS-Windows版本. 你可以解压下载的几个文档或是一个大一点的. 现在的很多电脑用户更喜欢第二种方法. 对于第一种办法需要:

List
```
- 编译好的Vim可执行文件
- Vim运行时文件
- 解压zip文件的程序
```

要得到Vim的文档，可以在镜像站点中找一个离你最近的，现在的镜像站点列表应该能帮你找到最快的：

ftp://ftp.vim.org/pub/vim/MIRRORS

或者访问从主站点ftp.vim.org，如果你觉得它已经够快的话．在"pc"目录下你会看到很多文件．版本号跟文件名连在一起．往往最想下载的总是最新版．

List

gvim61.exe	可自解压安装的文件.

对第二种方法来说只要这一个就够了．只要运行这个程序，照着提示做就行了．

对第一种方法来说你还要选择一个二进制文件．一共有下面几个可用：

List

gvim61.zip	普通的 MS-Windows GUI 版.
gvim61ole.zip	带OLE支持的 MS-Windows GUI 版. 吃掉更多内存，不过能支持与其它OLE程序交互
vim61w32.zip	32 位 MS-Windows 命令行版. 用于 Win NT/2000/XP 的控制台. 在Win 95/98下不行
vim61d32.zip	32 位 MS-DOS 版. 这个可在 Win 95/98 命令行窗口中使用.
vim61d16.zip	16 位 MS-DOS version. 只用在一些老系统上. 不支持长文件名

你只需要这些文件的其中一个．虽然你也可以同时安装一个GUI版和命令行版．运行时文件总是需要的．

List

vim61rt.zip	运行时文件

un-zip程序可以解压这些文件．如用"unzip"来解压：

shell command

```
cd c:\
unzip path\gvim61.zip
unzip path\vim61rt.zip
```

这将会把文件解压到目录"c:\vim\vim61"．如果你系统里哪个地方已经有了一个名为"vim"的目录,你需要转到它的上层目录再执行上面的操作．

现在可以到"vim\vim61"目录下运行安装程序：

shell command
```
install
```

注意看程序的提示信息以准确选择你要的选项. 如果你最终选择了"do it"安装程序就会执行你所选择的操作.

安装程序不会去动运行时文件. 它们还留在你把它们解压的地方.

如果你对预编译的可执行文件中带的功能不满意的话, 还可以自己编译. 首先弄到一份源代码. 然后找一个编译器, Microsoft Visual C就行, 不过太贵了. 可以用免费的Borland命令行编译器5.5版, 或者是MingW, Cygwin编译器. 文件src/INSTALLpc.txt里有相关的提示.

90.3 升级

如果你已经在运行Vim但想安装另一个版本, 本节就是讲述如何升级的.

UNIX

运行"make install"会把运行时文件复制到你指定的目录. 这样它就不会覆盖掉以前的版本了.

可执行程序"vim"则会覆盖它的老版本--不过如果你不在乎的话, "make install"也没意见. 你也可以手工删掉老版本的运行时文件. 只需整个删除带着版本号的那个目录. 如:

shell command
```
rm -rf /usr/local/share/vim/vim58
```

通常来说这个目录下的东西都不会被动到, 如果你动过了象"filetype.vim"这样的文件, 记得在删除它们之前把你自己喜欢的一些设置合并到新版本中去.

如果你只是小心翼翼地想尝试一下新版本感觉如何, 可以把新版本安装在另一个目录下. 这需要另定一个配置选项. 如:

shell command
```
./configure --with-vim-name=vim6
```

真正运行"make install"之前, 你也可以用"make -n install"预演一遍看该命令的执行会不会覆盖你要的东西[1].

如果你最终决定换用新版, 只需把可执行文件名改过来就行了, 如:

[1]译: make -n参数是只显示要运行的命令, 而不真正运行这些命令

shell command
```
mv /usr/local/bin/vim6 /usr/local/bin/vim
```

MS-WINDOWS

对MS-WINDOWS来说升级跟安装新版也差不多. 解压文件. 通常会生成一个新的目录名，如"vim61". 新版本的文件就存放在该目录下，你的运行时文件，vimrc配置文件，viminfo文件等等则留着不动.

如果你要运行新版，多少得多动一下手. 不要运行install程序，它会覆盖掉你旧版中的一些文件. 可以通过指定绝对路径来运行新版程序. 启动后它会自动找到相应版本的运行时文件. 不过，如果你把$VIMRUNTIME变量设错了可不行.

如果你对新版试用满意，就可以把旧版的文件给删了. 参考|90.5|.

90.4 常见问题

本节回答关于安装Vim的一些常见问题.

Q: 我没有Root权限该怎么安装Vim?(Unix)

用下面的配置命令，把Vim安装到$HOME/vim目录去:

shell command
```
./configure --prefix=$HOME
```

这会为你安装一个个人专用版. 你需要把$HOME/bin加到你的搜索路径中. 参考|install-home|.

Q: 我的Vim显示的颜色不对(Unix)

用下面的SHELL命令检查一下你的终端设置:

shell command
```
echo $TERM
```

如果列出的终端类型不对的话，给它设置一个恰当的值. 关于这个|06.2|处有更多的提示. 另一个解决办法是使用GUI版，这个不需要终端设置.

Q: 我的删除键和退格键不能正常工作

对<BS>和<Del>两个键来说，该送什么码并不十分固定，首先还是要检查$TERM的设置. 如果这个没问题，试试命令:

ex command
```
:set t_kb=^V<BS>
:set t_kD=^V<Del>
```

上面的命令中第一行你需要先按下CTRL-V然后按退格键. 第二行中你需要先按下CTRL-V然后按删除键. 你也可以把这两个命令放到你的配置文件中去，见|05.1|. 缺点是哪天你换用了另一种终端它就又不正常了. 可以参考|:fixdel| 试试另一种解决.

Q: 我用的是RedHat Linux. 我能使用系统自带的Vim吗?

默认情况下RedHat安装的是一个最小的简装版Vim. 检查一下系统中所安装的RPM包. 看有没有一个叫"Vim-enhanced-version.rpm", 你可以安装这个.

Q: 我如何打开语法高亮功能? 怎样使用plugins?

使用示例的vimrc配置脚本. 你可以参考|not-compatible|了解如何使用它.

第6章有关于语法高亮的信息: |usr_06.txt|.

Q: 什么样的vimrc配置文件好用?

参考www.vim.org网页上的一些例子.

Q: 在哪能找到Vim 插件?

Vim-online站点: http://vim.sf.net. 很多用户把一些有用的脚本上载到了这里.

Q: 在哪能找到更多的技巧提示?

Vim-online站点:

http://vim.sf.net

那有一个集结了所有技巧提示的文件. 你也可以搜索邮件列表文档|maillist-archive|.

90.5 卸载Vim

总有这样的情况: 你不得不卸载Vim. 本节讲述卸载.

UNIX

如果你是以软件包的形式安装的Vim，你应该看一下你的软件包管理工具如何卸载它.

如果你是从源代码编译过来的你可以用命令:

shell command
```
make uninstall
```

来卸载. 不过, 如果你已经删除了原来编译时的文件或者你用的是别人编译的, 你就不能用这个命令了. 手工删除好了, 下面是"/usr/local"用做顶层安装目录时要删除的文件列表:

List
```
rm -rf /usr/local/share/vim/vim61
rm /usr/local/bin/eview
rm /usr/local/bin/evim
rm /usr/local/bin/ex
rm /usr/local/bin/gview
rm /usr/local/bin/gvim
rm /usr/local/bin/gvim
rm /usr/local/bin/gvimdiff
rm /usr/local/bin/rgview
rm /usr/local/bin/rgvim
rm /usr/local/bin/rview
rm /usr/local/bin/rvim
rm /usr/local/bin/rvim
rm /usr/local/bin/view
rm /usr/local/bin/vim
rm /usr/local/bin/vimdiff
rm /usr/local/bin/vimtutor
rm /usr/local/bin/xxd
rm /usr/local/man/man1/eview.1
rm /usr/local/man/man1/evim.1
rm /usr/local/man/man1/ex.1
rm /usr/local/man/man1/gview.1
rm /usr/local/man/man1/gvim.1
rm /usr/local/man/man1/gvimdiff.1
rm /usr/local/man/man1/rgview.1
rm /usr/local/man/man1/rgvim.1
rm /usr/local/man/man1/rview.1
rm /usr/local/man/man1/rvim.1
rm /usr/local/man/man1/view.1
rm /usr/local/man/man1/vim.1
rm /usr/local/man/man1/vimdiff.1
rm /usr/local/man/man1/vimtutor.1
rm /usr/local/man/man1/xxd.1
```

MS-WINDOWS

如果你是从那个自解压文件中安装的Vim, 你也可以运行Vim可执行程序所在的目录下的"uninstall-gui"程序来卸载. 你可以从开始菜单中选取它来运行. 运行这个卸载程序会移除大部分文件, 快捷菜单中的菜单项和桌面上的快捷方式. 还有一些文件需要Windows下次启动时才会被删掉.

卸载程序会问你是不是要整个删除"vim"目录. 这个目录通常也是你的vimrc配置文件所在. 所以小心一点.

或者，如果你是从几个zip压缩文件里安装的Vim，最好的办法是运行"uninstal"程序(注意我没有少拼一个l). 它跟"install"程序在同一个目录中. 典型地: "c:\vim\vim61". 这也可以从控制面板里的"安装/卸载"程序里卸载.

不过这个卸载办法只是删除了Vim相关的注册表项. 你还是要手工删除相关文件. 选择"vim\vim61"目录整个删除即可. 如果你改了这个目录下的什么文件，最好还是先检查一下.

"vim"目录里可能有你的vimrc配置文件和你创建的其它一些运行时文件. 看一下你是不是要留住它们.

目录: |usr_toc.txt|
版权: 请参考|manual-copyright| vim:tw=78:ts=8:ft=help:norl:

Vim作者专访

本文原文位于http://web.efrei.fr/aiefrei/effervescence/123/vim.en.html

记:Sven，Bram，首先感谢两位接受这次专访，你们先自我介绍一下好吗?

Bram:我出生于1961年，我是一个专业的计算机技师。在Delft技术大学修完电子学之后我的大部分工作都在研究海洋生物颜色试验的复制和打印设备。我使用计算机已有相当长的时间了。我在市场上还在大卖4K的RAM时建造了我的第一台计算机。后来我从焊接转到了编程上来，现在我还是很喜欢电子学，只是没有象以前那样花费那么多的时间了。

目前我住在Venlo，荷兰的一个小城市，工作之余我喜欢听音乐。麦克风是我的最爱。但除此之外我没有其他的业余爱好了，也不需要托家带口，所以我有很充裕的时间花在编程上面，我喜欢编程。每年我大概旅游一两次，通常是到一个文化背景完全陌生的国家去，这样可以了解世界上不同地方的人们是怎样生活的，也可以拓宽自己的视野。

Sven:我1967年生于德国柏林。我一直在柏林生活，我目前在从事数学和计算机科学的研究。1989年我通过Email了解到Internet，1992年我开始使用新闻组，我经常用它进行通讯。Web出现后我开始在网页上整理我过去使用过的一些软件

我曾经玩过吉它，不过自从开始维护这些程序(elm，lynx，mutt，nn，screen，slrn，Vim，zsh)的网页后，也顾不上它了。这些程序都有一个共同点: 它们都有一个字符界面。所以它们可以在十分普通的字符终端上运行。只要能通过telnet进行互接就行(如果你愿意，也可以使用ssh)。

1999年9月我终于去了美国，那次我是到加利福尼亚旅游，在那里我见到了我认识已久的网友们。

我从没见到Bram本人，所以我真得看一看他的照片才知道他长什么样。

记:你什么时侯开始写Vim，是什么促使你想写一个这样的软件，能跟我聊聊Vim的发展史吗，可以说Vim是最好的文本编辑软件吗?

Bram:我在1989年开始写Vim，那时我刚买了一台自己的计算机，Amiga2000那时我用的是vi，我想用一个与vi类似的更好一点的编辑器，能找到的几个vi的克隆版都不够称心，所以我用了Stevie的源代码，开始一点一点地修改，添加一些命令。

第一个可用的版本很快就出来了，1991年发布，这一下招来了很多人的回应，他们鼓励我把它再增补一些功能，也就是从那时''Vi Imitation''更名为''Vi IMproved''，这之后我就一直往上面添加新的功能。

Sven:1992年我就在找一个可以在我的Macintosh IIvx机上用的vi编辑器。后来我又回过头来用了一阵子DOS(原因很明显)，虽然我有更多的

理由喜欢Macintosh，但没有Macintosh版的vi。

1994年有人向我推荐了Vim--2，我惊喜于终于有人开始增补Vi的功能了。从那时起我就开始通过一个主页支持Vim的发展，同时希望有一天它能被移植到Macintosh上去。

Vim--3版终于被移植到Macintosh上时我高兴坏了，不过那个版本有很多BUG，还有几个很要命的问题。

Vim--4版没有出Macintosh的版本，这实在让我失望。

后来，Linux发展得如火如荼，我又转向了PC。

1997年9月我注册了域名Vim.org，可以更好地支持Vim的发展了。

今天的Vim已经远非昔比了，很多其它编辑器的优良特性也应用户之需添加到Vim里，Vim几乎可以运行在每一种平台上，又有了图形界面。不过Macintosh版的移植问题还是让我...

记:你平均每周花多少时间在Vim/Vim网页上维护上，花多少时间去回EMAIL ，回覆comp.editors新闻组，你怎样安排花费在你的工作任务和Vim上的时间?

Bram:我花在阅读和回覆Vim新闻列表上的时间太多了，有时侯我真想不管它了，但我看不得那些邮件没人回覆。现在还可以，工作之外我的时间还很充裕，不过我还是经常在半夜里才有时间回那些邮件。

我决定不在comp.editors上活动了，太浪费时间了，幸好有Sven和其他一些人及时回答人们提的问题。

我几乎每天都要为Vim花点时间，有时只是修改一点小问题，有时就要大刀阔斧地做一些大动作了。只有我不在家时我才可能一整天不碰Vim。

要是我找了份别的工作，就很难抽空来做这些了，恐怕只有周末才有时间，也正是因为这个我才没去找工作。

Sven:我几乎所有的时间都花在Vim上，大部分时间都花在回答问题(comp.editors上的贴子，直接发给我的邮件，或者是发在Vim的邮件列表上的邮件。)和维护www.Vim.org网站上了。

目前为止我还很少在德国的Vim站点上谈论Vim，但经常跟一些已经熟知Vim的人说起过。我希望能尽力为大众读者写一点文档。网页上已经有了一些这样片断的帮助文档。但写这些东西实在太花时间了。

每每如此，我都希望能创建一个全面的帮助文档数据库，内容包括过去的VI文档，邮件列表，新闻组和其它尽可能多的内容。以一种色调适当可读性较强的形式展现给读者。但，仍是老问题，这不知道要多久...

记:在VIM的编程方面，由谁来开发代码，一共有多少程序员，谁来协调项目的发展，你通常是如何决定各个版本的发行日期的? 你是如何做

到让来自分散开发的代码协同起来的? 是不是任何人都可以加入开发小组? 在对待新的函数库如GTK+图形库方面，你如何决定取舍?

Bram: 我自己负责大部分核心代码，但长期以来也有其他几个主要的志愿者帮助开发。图形用户界面部分由Robert Webb开发，其它很多模块由别人来开发，通常这些人都是在上面短期开发一阵子，然后就中止了，我要使这些断断续续的工作能得到持续的进展，保证整个项目有一个正确的发展方向。此外，还要维护其它小组成员的代码，由于Vim运行在如此多的平台上，这实在是一件苦差。

经常是有人发送给我一个补丁包，由我来决定要不要把它加进去，其实很难拒绝别人的补丁包，因为这都反映了用户本人急切希望看到的特性。但同时也有人抱怨说Vim变得越来越大了。我要做的就是尽量在这两者之间保持平衡。

通常Vim的开发问题我说了算。先是倾听人们对Vim的期望，然后在用户需求的基础上做出决定。有时用户所要求的东西和他们真正需要的东西往往不是一回事，这时就要细心分析他们所提出的要求，发现这背后隐藏着的真正的用户需求。或许有些东西在编程实现起来就没那么容易，这也促使我一直都在使用Vim，这样我才能亲身体会用户对它的期望和要求。

原则上来说，每个人都可以把补丁发给我。如果我觉得有必要就会把它加入到Vim的正式版本中来，这也看发补丁的人要求的是什么东西，另外，补丁本身的质量也要慎重考虑。但是Vim已经变得很庞大了，所以我更要细心甄选。

GTK库的移植已经被加入，因为它运行得不错，甚至比Athena或Motif都要好，另外，它还是自由软件。头痛的是最初的移植者没有时间去修改BUG，这样我们就要花力气维护它的稳定性，好在现在已经运行得相当不错了。下次再有这样大的改动我可得加倍慎重了。因为加入一个不能运行的特性有害无益。

Sven: 现在Vim是众多程序员心血的结晶。但Bram起到了决定性的作用。Bram的工作十分出色。毋庸置疑，Bram是一个经验丰富的程序员，并且他一直都在使用这个程序，这使得他对Vim了如指掌。

我实在佩服他的责任心和对各种争端的把握能力---即使是一场激烈的争论。我衷心希望他能为自己的努力得到应有的报酬，好让他能象现在这样细心呵护Vim的成长。

但这对开源软件的贡献者们来说还是一件没影的事---到哪里去找这样的报酬呢? 也许象linuxfind.org这样的站点会提供一些赞助。

记: 现在的Vim代码可以工作在各种平台上，它的体系结构是如何设计的? 能否让它很容易地运行在一种新的平台上?

Bram: Vim的体系结构一直在不断的革新之中。有时要加入一个新的特性不需要改动太多的代码。如果新加入的特性打乱了整个项目，那我就要费一番心思好好想想办法了。现在的Vim是一个很大的程序。这样做可不容易。

每一次修改都可能引入新的BUG，每不幸，这说明我们当前的体系结构并不是十分理想。如果有时间我可能在6.0版中做很多的改动。

增加一个新平台的支持应该不会太难。如果你有这样一个平台，不妨拿来试试看。很多UNIX类平台都可以一试，但是象VMS这样的系统就很难支持了。Vim 在UNIX上的背景并不特别适用于VMS。实际上Vim真正的出生地是Amiga，但这对UNIX平台的移植来说不是难事。

Sven: 开发小组的程序员们都知道Vim的源代码十分出色。但是每增加一个新的特性都会增加系统的复杂性。这就使得很难为此写出补丁。

每次我欣喜于Vim又加入了新GUI特性而更受欢迎时，这些新的GUI特性都会消耗一大堆补丁。采用一种新的GUI总是特别费事，当然开发起来也更费时间。

有时甚至增加一种新平台的支持都比增加另外一个GUI来得容易...

但是不管怎么来说Vim的发展还是十分迅速的---相对于跟不上步的文档来说。有必要增加更多的文档来说明如何充分地使用Vim。

记: Vim只能用于英语。有没有让它国际化的方案?

Bram: 程序本身并没有计划要国际化。那样太艰维护了。光是翻译这些文档就够呛了。但是现在正在计划让它可以编辑多种编码方案的文本。

现在已经可以支持希波莱文和波斯文以及一些亚洲语系的编码了。对我们来说，从右到左进行阅读，看着新插入的字符是往后倒的真是觉得好笑。在6.0 版本中我们计划要加入UTF-8格式(什么东东?)，现在这种格式的使用远比以前要来得普遍了。

Sven: 是的，增加对UTF-8的支持是一件很有意思的事，这个目标应该在不久就能实现。

我希望Vim会加入国际化的消息显示，而且最好有人来翻译在线帮助系统。并且能及时跟踪变化的步伐。(暂不考虑从ASCII字符到其它字符集带来的问题)

我们已经在为帮助文档的翻译做出努力了。欢迎大家踊跃支持---但是这是一项艰苦的任务。

记: 你在Vim上所做的工作有没有给你带来新的工作机会? Vim如何影响你的生活?

Bram: 目前为止还没有人因为Vim给我提供一份工作。但是我的确与世界各地的人们保持着联系。也许这会有助于我找到一份不错的工作吧。

与世界各地的人们保持联系是一件很有趣的事。虽然只是通过e-mail。各种不同的文化就隐藏在他们的字里行间里。e-mail让这一切都成为了可能，这真是太不可思议了。刚开始使用不觉得怎么样。但它能让人们空前地聚集在一起。

Sven: 跟bram一样---我也没有因为Vim得到过工作。不过，我用EMAIL与人们说起我的工作时，他们都会说那不成问题---特别是如果我能到美国的话。

很多用户说他们希望他们的程序能支持一种外部的编辑器这样他们就可以使用Vim。我多么希望有这样的公司能采纳这样的意见。

谁知道---也许有一天你能在一个带有外挂编辑器的程序内用Vim编辑文本。

记: Vim是慈善软件，这就是说人人都可以自由地使用它，但是鼓励大家为乌干达的孤儿们做出善举。能详细谈谈这个吗?

Bram: 这事要从我自己的故事说起。

我已经很多次利用假期的时间到不同的国家与当地的人们呆在一起。这种方法让我了解更多的人们。

我于1993年在乌干达呆了一个月。那是个很特别的地方，因为这是我们第一次碰到切切实实需要我们帮助的人们。在其它地方人们并不会真正需要我们去帮助他们。或者我们没办法去怎么帮助他们。但在乌干达，那里的人们确实需要帮助。三周的时间内我们成立了一所幼儿园。那是一次很特别的经历。

第二年我又去了那里一个月。看到那里的孩子们在新的学校里上学。这种情景大大地触动了我，我决定在那里工作上一年。一年时间里我致力于提高当地人们的饮水与医疗设施的水平上。也更多地了解了当地的人们。

回家后，我想继续帮助他们，于是我在荷兰成立了ICCF基金会。把我在Vim上的工作跟这个结合在一块。我发现这的确发挥了效应。因为Vim的原因，这个项目有了更多的捐赠品。这起到了十分积极的作用。

所有收到的捐赠品都原封不动地用在了孩子们身上。我们的目标是争取有5%的人来捐赠，实际水平还低于这个目标(1998年是1.9%)。

Sven: 我第一次通过EMAIL与Bram接触时他就在乌干达。那里我们之间多数联系都要通过中间人。发送EMAIL可不象现在这样发到‘‘user@domain’’这么简单。

当我读到‘‘Vim是慈善软件’’时我意识到Vim不仅帮助了它的软件用户们，也帮助了很多孤儿。所以我希望我在Vim上的努力也会对他们起到间接的帮助作用。

我没去过乌干达，但我在1977年住在东非肯尼亚的几个月时间里接触了那里的人们。也去学校看了看(在kissi---kissi是什么地方)。

每次看到从外面公司来的EMAIL我都宁愿他们问能否为乌干达的孤儿们做点什么而不是问Vim的千年虫问题。

记：世界个有多少人使用Vim？ 他们需要什么？ 为什么他们选择了Vim？ 他们主要的需求是什么？

Bram: 很难说到底有多少人在使用Vim，因为Vim不需要注册。但大致看来。每个linux的分发包里都包含一份Vim。有人广泛征问，得到是估计有5%的Linux 用户经常使用Vim。Linux的用户大概有1700万。那就是说大概有85万的Vim用户。再加上在Solaris或Windows等平台上使用Vim的用户。

Sven: 的确是，Vim不需注册。这很好，你不希望一个自由软件还要注册，对吧？ 任何形式的注册要求都可能会赶跑Vim的用户。

我恐怕难以估计Vim的用户数，计算这个数字只会浪费时间。

不过有一件事可以肯定---越来越多的人们从最初的vi转而使用vi的各种克隆版本(elvis，lemmy，nvi，vile，Vim)。

Vi的克隆版本在功能上的增强是人们纷纷转向他们的根本原因。至于Vim 的用户，每一次新功能的增加都为用户转向它增加一个砝码。

Vim-5最大的新增特性是语法高亮，这一功能使得Vim可以根据当前编辑的文本类型所对应的语法规则以不同的颜色显示文本的不同成分。通常这用于程序员们编辑他们的源代码文件。不过Vim也能很好地支持EMAIL和新闻组消息的语法高亮功能。

记：Vim5.4版刚发布你就张罗着要在2000年发布Vim6的事了...下一个千年中你对Vim有什么计划？

Bram: 关于这个问题你最好看一下Vim的TODO列表。鉴于TODO列表现在的份量，我可能要用接下来的整个一千年都用在Vim上才行！ :-)

去年进行了一次关于用户最希望的Vim功能的投票。这有助于我决定接下来要做什么。其中一个特性是'折行'。这是Vim6.0中首要的新增特性。'折行'可以隐藏文本的一部分，这样可以很容易把握整个文档的结构。比如说，你可以把函数体都给'折叠'起来。这样你就可以对函数有个大体的把握，或许你要重新排列它们的顺序。光是这个就有很多具体的事要去做，所以最好为此发布一个新的版本。

Sven: 1998年的投票结果显示'折行'是用户最希望的特性。所以Vim-6的主要目标就是加上这一特性。

其次，用户最希望看到的特性是'垂直窗口分隔'，这一特性可以创建垂直显示的子窗口。因为很多用户希望能比较两个在内容上相近的文件的不同之处。特别是他们在编程的时侯。

用户呼声排名第三的是为各种语言量身定做'可配置的自动缩进'，因为Vim首先是一个程序员的编辑器...

但是并不是说你非要是一个程序员才可以用好Vim。事实上，我更希望Vim能为初学者增加一些在线支持好让他们熟悉'模式编辑'的概念，但这无疑会增大可执行文件的大小。所以我只能寄希望于帮助文档了。

虽然Vim尽力保持与Vi的兼容性，但可以肯定大多数用户还是很喜欢Vim 的增强特性，特别是看到屏幕上可以看到关于当前编辑信息的提示，文件名和缓冲区号，光标位置和其它一些选项如最大行宽。

对初学者而言能在屏幕上看到所有的字符是很重要的(行尾空白字符和特殊字符)。特殊颜色的显示解决了这个问题。Vim内置的文档格式化功能对于要发送到新闻组的文本或EMAIL信件来说是一个十分强大的工具。

通过网络连接编辑一个远程文件，'折行'(见上)，使用多字节字符语言的支持也是得票很高的功能。但是这些做起来可不象说的那么容易。同时我也希望大家能帮助我们实现这些诱人的功能。

我尤其希望看到'折行'功能。很多用户将借助这一特性浏览他们文件的大纲。邮件列表上的讨论表明实现这一点还有很多困难要克服，这真是太富有挑战性了!

不过，我还是认为在增加这些重头戏之前先解决一些小问题。(比如:一个内置的非图形用户界面的文件浏览器。)

TODO列表是一个技术上的目标清单。上面的很多条目标记着一些技术上的目标。如果你不能熟知Vim的所有概念可是很难看懂这个列表的。

很多特性的增加是为了弥补缺乏图形用户界面的不足。而且，很多使用Vi运指如飞的用户根本不需要一个图形用户界面。我本人使用GUI也只是为了帮助了解别人关于这方面的问题，做自己的事我从来都不用它。

从上面的这些你也看得出来我的兴趣主要在于Vim的非图形用户界面的特性上，这样的特性可以在的终端上使用，只要用命令本身就行了。我所收到的反馈也表明大多数人要的正是这个---而不是诸如菜单或一些特殊程序的支持如拼写检查器或编程语言方面。

所以很久以前我就有了一份这样的列表，描述用户在每天的编辑过程中实际碰到的问题，以帮助开发小组认识到这些问题并提出解决方案。

记: Sven, Bram,再次感谢你们的帮助。同时恭喜你们在Vim上所做出的出色的成果。还有什么要补充的吗?

Bran: 谢谢你的专访。商业程序可以用广告来吸引用户，象Vim这样的自由软件就要靠别的办法了，所以感谢你这样的专访。

我十分喜欢用Vim进行编辑，我希望我在开源软件上的努力也能帮助更多的人们达到他们的目标。

如果你使用Vim过程中遇到问题可以试一下``:help''，所有的在线帮助文档都是纯文本的，所以打印出来应该不会有什么问题。

附加的文档也补上了，比如德语版的HowTo和``Vim初学者''(也是德语)。

你也可以在comp.editors提问，或者通过6个专用邮件列表---通用的帮助列表。

欢迎在你的主页上发布你的个人使用技巧或窍门。或者是维护一个语法文件，某种操作系统的二进制文件，或者在新闻组，邮件列表里回答别人的问题

非常非常欢迎提供各种形式的帮助。:-)

interview by Herve FOUCHER

七个有效的文本编辑习惯

作者Bram Moolenaar
Bram@Moolenaar.net
slimzhao@hotmail.com 翻译整理

如果你要花大量的时间键入文本，写程序或编写HTML脚本，你可以通过有效地使用一个好的编辑器来替你节省时间。本文将引导你如果快速地完成你的编辑工作，并且减少你的错误。

本文将以开放源码软件Vim(Vi IMproved)为例向你展示如何进行有效的编辑，但这里提到的原则对其它的编辑器也是一样，选择合适的编辑器是进行高效的编辑的第一步，关于哪个编辑器最好的争论已经数不胜数，本文不打算对此再说些什么。如果你还不知道用什么编辑器或者觉得你现在使用的编辑差强人意，试一下Vim，保你满意。

第一部分： 编辑一个文件

1. 快速移动

文本编辑的多数时间都花费在浏览，检查错误或者找出你要进行编辑工作的正确位置，输入新的内容或改变已有的内容倒在其次。在文本中随意漫游是非常常见的操作。所以高效编辑的第一要义是学习如何能够在文本中快速移动，准确定位。

通常情况下，你知道要查找的内容，或者查看所有的文本行只是为了找出某个单词或者短语。你可以使用查找命令"/pattern"查找文本，但有几点要注意的:

如果你已经找到了一个单词并且想找出这个单词还在其它哪些地方出现，可以使用"*"命令，它查找下一个匹配的目标。如果你设置了'incsearch'选项，Vim将会以反白显示出第一个被找出的匹配。这能在你还在/命令下敲入关键字时就快速地显示出来(类似于emacs的递增查找功能) 如果你设置了'hlsearch'选项，Vim将会高亮显示所有查找到的匹配，这种策略可以让你对要查找的内容有一个概括的了解，如果你在程序代码中使用这一功能，它能显示出所有引用某个变量的地方。你不需要移动光标就可以看到所有符合条件的匹配(同一屏幕上可以看到不至一个地方被匹配)。

在一些结构规范的文本中还有其它一些更方便的小技巧进行快速移动，Vim内嵌了方便C程序(以及与C语言很相象的C++和Java)的命令： 使用"%"命令可以从一个打开的括号跳转到与它成对匹配的另一个括号处，还可以从一个预处理指令"#if"跳转到与之匹对的"#endif"。其实"%"命令能跳转到好几种文本元素的'另一半'去。这对检查你的() 和{}是否正确匹对非常方便。使用"{[{"跳转到当前代码块的开头(代码块是用"{}"括起来的程序段)。

使用"gd"可以跳转到当前光标所在的单词(变量)的局部定义处。当然，还有很多其它的技巧。关键是你要知道有这样的命令。你也许会说你不可能学习所有的命令---共有几百个不同的移动命令，一些很简单，还有一些是智能化的---不过它可能要花费你数周的时间学习使用它们。当然，你不必全部掌握，只要有你自己的一套办法，并且能处理你所要进行的操作。

有三个步骤可以使你学到你需要的技巧:

当你编辑文件的时侯，留意一下你经常要重复进行的操作是什么。或者你花大部分时间都在干些什么。想一想有没有一个编辑命令可以替你做最让你头痛的事。读在线文档，问一个朋友，或者看一下别人是怎么做的。

练习使用这些命令，直到你的手指可以不假思索地运用自如。举个例子来说明到底怎样做:

你在写C程序的时侯，你经常要花时间找到一个函数的定义。现在你使用的是"*"命令查找这个函数名都在哪些地方出现过，但在你到达真正的目标之前，可能还有符合你的查找条件的很多个匹配(如注释中出现的或该函数在其它地方被调用）骚扰你。你可能会想一定有一种捷径可以一步到位。

浏览一下参考手册你就会发现关于tag的主题。文档会告诉你如何使用这一功能跳转到函数的定义处。这正是你要的东东!

你已经知道如何生成一个tags文件(ctags *.[ch]或etags *.[ch])，使用ctags 程序就可生成Vim所要的tags文件。接下来你练习使用CTRL-]命令。为了更方便地使用这一功能，你还可以往你的makefile文件里加入自动生成tags文件的命令。当你使用这面的三个原则时要当心:

"我想使用这些命令，但我没时间去看文档中的一些新命令"。如果你还这样想，那么你可能还处于计算机的石器时代(就是说你比较菜啦)。有些人做什么都用notepad，他们可能觉得别人用更短的时间完成相同的工作是不可思议的事。

不要重复做相同的事。如果你经常要去找一个你常用的命令，你就没时间专注于你手头上的事的。只要找到耗费你太多时间的操作，练习使用这些操作对应的快捷命令，直到你可以不假思索地使用它们。这样你才可能把精力集中在你要编辑的文本上面。

下面是一些多数人都会遇到的常见问题的解决方案的建议。你可以以此为例，学习使用上面的三个原则。

2. 不要两次键入同样的东西

我们键入的文本都是一个有限的集合。甚至使用了有限的短语和句子。尤其是计算机程序。显然，你不必两次键入这些相同的东西。

最常见的事是你要把一个词改为另一个，如果你要将整个文件里所有地方出现的这个词都换为另一个，你可以考虑使用":s"命令，如果你要有选择地进行更改，而且最好在看了上下文之后再决定，你可以使用"*" 命令查找这个词的另一个匹配，如果你决定要改，那么使用"cw"使用改变这些词，然后再用"n"命令到下一个匹配处使用"."重复上一个命令。"."命令重复上一次改变。一个改变，是指插入或删除或替换一些文本。可以对这些操作进行重复是一种功能强大的机制。如果你用它来组织你的编辑操作，很多以往必需手工做的修改就只需要简单地使用"."命令。要特别注意在重复上一次修改操作之前你有没有做其它事，夹在中间的有些操作可能会改变"."命令实际重复的内容。使用"m"命令标注文本的一个位置地很有用。它可以让你在作了重复的修改之后回到你上次停留的地方。

一些函数名和变量名很难正确地键入，比如"XpmCreatePixmapFromData"，没有一个样本看着或不看它的帮助是很难的(至少是很烦的)。Vim有一个补全机制可以让这种事变成小菜一碟。它会在文件里查找你要键入的文本，找到相近的匹配就直接插入，而且，它还在你的include文件里递归查找。你可以键入"XpmCr"，接着按下CTRL-N键，Vim会把它扩充为"XpmCreatePixmapFromData"，这样的功能还来的不光是为你节省了时间，它还减少了你手工键入时出错的机会，而且，你的编译器也不会产生那么的警告错误了。

如果你要重复键入一个短语或一个句子，也有一种快捷的方法。Vim有一种记录宏的机制。你键入"qa"开始把一段宏记录入寄存器变量‘a’中。按下来你可以象平常一样键入你要的操作，只是这些操作都会被Vim记录进它命名为‘a’的宏中，再次再下"q"键，就结束了宏‘a’的录制。当你要重复执行你刚才记录的那些操作时只要使用"@a"命令。共有26个可用的寄存器供你记录宏。

使用宏你可以重复多个不同的操作。而不仅仅是插入文本了。如果你要进行某种重复的操作，记着要用这一招呀。

使用宏要注意宏只是机械地重复你刚才键入的动作，当你在文件里移动时要小心。你用宏重复时和你当初录制时要操作的文本对象可能不一样。你录制宏时向右移4 个字符可能对它当前的环境来说是正常工作。但当你回放这些宏时，它工作的文本环境可能需要移动5个字符。

当你要录制的操作比较复杂时，要想一次就全部通过也不是一件容易的事，此时你可以写一段宏或脚本。这对于使你的程序模板化非常有用。比如，一个函数头，你可以把这项功能定制得如你所愿的智能化。

3. 错误修复

打字时出现错误是在所难免的事，办法只有一个，就是尽快纠正它。编辑器可以帮你自动做这一工作。但是你要事先告诉它怎么才算错，正确的又是什么。

对常人来说，常犯的错误都是同一个错误。你的手指就是不听使唤。这可以通过缩写功能来纠正。一些例子是:

```
:abbr Lnuix Linux
:abbr accross across
:abbr hte the
```

你一键入完错误的词编辑器就会用正确的词来替代它。

同样的机制也可被用来以少数几个字符代替键入一个长的词。特别是一些你很难正确拼写出来的词。这样也避免了你犯错误的机会。例:

```
:abbr pn pinguin
:abbr MS Mandrake Software
```

不过，副作用就是编辑器总是试图把它所知道的缩写扩展为整个单词，如果你真想键入MS，反倒成了一个难题。所以尽量使用没有歧义的缩写。

Vim有一套优秀的语法高亮机制找到你的文本中存在的错误。程序员尤其是这一功能的最大受益人。

语法高亮用特殊的颜色来显示注释。这听起来好象没什么，但一旦你使用了这项功能你就会发现好处多多。你可以快速发现哪些部分应该是一个注释。但是并没有被语法高亮指出来。对程序员来说，忘记注释的结束标记*/是很正常的事。这在只有黑白两色的文本中可不是一件省油的事。

没有正确匹对的括号也可被语法高亮指出。一个没有被正确匹对的括号")"会被一个亮红色的背景特别指出。你可以使用"%"命令看一看它应该跟谁匹配，然后在正确的位置补上一个"("或")"

其它的一些常见错误也可被语法高亮功能协助你检查出来，如#included <stdio.h>。在黑与白的世界中它们对错难分。但语法高亮可以帮你快速分辨出雌雄真假。

一个更复杂的例子: 对于英语文本来说，可以有一个长长的可用单词的列表，不包括在其中的单词都被视为一个错误，使用一个语法文件，你可以把所有没有出现在该文件列表中的单词用语法高亮功能标出来。用一个特殊的宏你就可以往这个单词清单里加入新的生词。加入后它们就不再被视为一个错误了。这种功能以往只能在单词分析器中。在Vim中使用简单的脚本就可实现，而且，你可以按自己的需要来定制这一功能。比如，你可以只检查程序中的注释。

第二部分: 编辑多个文件

4. 经常需要编辑不止一个文件

人们往往都不是只编辑一个文件。通常有多个相关的文件。可能要在单个地编辑文件后一次编辑几个文件。或者同时编辑几个文件。要进行高效的编辑就要充分利用编辑器一次编辑多个文件的功能。

前面提到的tag机制可被用于在多个文件间跳转。通常的方法是为你正在做的项目生成一个tag文件。之后就可以在这个项目的多个文件之间自由跳转，发现函数定义，结构，类型定义typedef，等等。比起你单个

地搜索这些文件，可以大大节省你的时间；浏览一个项目之前第一要作的事就是为它创建一个tags文件。

另一个强大的机制是在一个项目中找出一个名字在多个文件中的不同地方，使用":grep"命令。Vim产生所有匹配的清单，并且跳转到第一个匹配处。"cn"命令可以使你跳转到它的下一个匹配处。这对于你要改变一个函数的参数来说非常有用。

被#include包含的文件含有丰富的信息，但是要找出你想要的东西却要耗费大量的时间。Vim可以处理#include所包含的文件。并且可以在其中查找你要找的东西。经常的需求是查看一个函数的原型。将光标定位在你要查看其原型的函数名上，然后按下"[I"命令，Vim将会显示include文件中匹配这个函数名的一个清单。如果你要看它的上下文信息，可以跳转到它的声明处。一个简单的命令可以用来检查你是否包含了正确的头文件。

Vim中可以把一个文本区分为几个不同的部分，然后分别编辑各个部分，编辑完成后你可以比较两个或多个文件的内容，或在它们之间copy/paste文本内容。有很多命令可以打开或关闭窗口，或在它们之间跳转。临时地隐藏文件。等等。再用上面的三个法则来练习你要掌握的新的命令。

多个窗口有多种用途。预览标签机制是一个很好的例证。它会打开一个特殊的预览窗口，并且使光标仍然停留在你当前所在的位置。在预览窗口中的文本列出了当前光标所在处的函数的声明(有些可能不是声明) 将当前光标移动到另一个函数名上，停留几秒钟，预览窗口中的内容就会变成是关于新函数名的声明。

5. 协同作业

编辑器是用来编辑文本的，e-mail程序是用来收发email的，操作系统是用来运行用户程序的。每个程序都有它自己的业务范围。将这些程序的功能组合起来就可产生强大的处理能力。

一个简例： 在一个清单中选择一些结构化的文本并且将它排序"!sort"。外部程序"sort"处理真正的排序工作。就这么简单，排序功能可以被集成进一个编辑器中。但是，如果你看一个"man sort"，你就会发现它有众多可用的选项。它有一个高度优化的算法来执行排序工作。你难道要在你的编辑器里写一个同样强大的排序程序吗？ 或者其它的流过滤程序？那将会使你的编辑器变得十分臃肿。

Unix的哲学是使用独立的小程序，每个小程序做一项专门的任务，并且把它作好，将它们的工作整合到一起来完成一个复杂的任务。不幸的是，多数编辑器并不能与其它程序一起协同工作，比如你不能替换Netscape里的e-mail编辑器。另一种做法是把所有的功能都包括到一个程序中去。在编辑器领域，emacs是这方面的一个典范(有人甚至说它是一个能编辑文本的操作系统)

Vim的做法是将这些分散的小程序整合起来，但这样做也并不容易，目前来说可以在MS的Developer Studio和Sniff中使用Vim编辑器，一些e-mail程序也支持外挂的编辑器，象Mutt，就可以使用Vim。与Sun的Workshop集成也可以正常工作。在这方面Vim还有待在将来进一步提高。直到我们找到一个比所有这些加起来还好的系统。

6.文本是结构化的

可能你经常要打交道的文本都有一些内在的结构。只是不被当前可用的命令所支持而已，你可能不得不要回头建立你自己的宏和脚本来操作这些文本。这样做显然有些复杂。

最简单的一件事就是加速你的编辑-编译-修改的周期。Vim有它自己的":make" 命令，该命令编译你的程序项目，捕获编译的错误/警告并允许你直接跳转到引起这一错误/警告的程序行上去。如果你有一个另类的编译器，它输出的错误信息可能对Vim来说是不可识别的。不要紧，更改你的'errorformat'选项，这一选项告诉Vim你的编译器将生成何种格式的错误信息，以便于它能识别。比如如何找到出错的文件名，出错的行号，既然它已经能与gcc产生的复杂的错误信息格式一同工作，可以想见，它也对付多数其它编译器产生的错误信息。

有时为一种特殊格式的文件作出调整也只是设置一些选项，写一些宏，如要跳转到manual 帮助文档，你可以写一个宏来获取当前当前所在的词，清除当前的缓冲区并且读入相应的帮助页，这对于查看交叉索引是一种简捷有效的办法。

使用上面的三项原则你就可以对付任何形式的结构化文本。只要想一下你要对文件做些什么，找出编辑命令，练习使用它。就象听起来一样简单。唯一的事就是你必须真正去做它。

第三部分

7. 养成习惯

学习驾车当然要花费心思，但这足以成为你继续骑自行车的理由吗?不，你意识到你需要投入时间学习一项技巧。文本编辑与此同理。你需要学习新的命令和技巧。

另一方面，你也不必学习一个编辑器所提供的所有命令。那样只会浪费你的时间。绝大多数人只需要学习其中的10-20%的命令就足以应付它们的工作了。但是对每个人来说，适合自己的命令集各各不同，这需要你不时地回顾以往所做的事，看看是不是可以自动完成一些重复的工作。如果你只进行了一次某项特殊的操作，并且没指望将来还要进行类似的操作，就不要试着去琢磨它了。但是，你也许能预见到在几个小时以内你就要重

复进行同样的操作。那么去文档里面搜索出你希望的"瑞士军刀" 或者要写一个宏来完成它。如果任务过于复杂，比如处理特殊类型的文本，你可以到新闻组里看看是不是已经有人解决了与你相似的问题。

决定性的步骤是最后一步，可能你发现了一个重复操作的解决方案，几个星期后你却又忘记了。那样没用。你要不断地重复练习你的解决方案直到你的手指可以条件反射地自动完成，从而达到你所期望的境界。不要一次尝试太多的东西，一次做一件事并多做几次会好得多。对于不经常的操作，最好记下你的处理步骤以备将来不时之需。不管怎样，只要目标明确。你就能找到让你的编辑变得更加高效的办法。

最后要提醒你的一点是人们往往还是会对上面提及的建议视而不见:我还是经常看到人们花费半天的时间在屏幕上用两个手指上滚下翻。真替他们感到费劲。用十个指头操作也并不会让他们更快一点，而且这样做也最容易让人心生厌烦。每天使用一个计算机程序一个小时，也只需要几个星期的时间练习这样的操作。

结束语

本文的由来是受Stephen R·Covey的名作"The 7 habits of highly effective people"启发。我向我知道的每个人推荐它去解决个人的或专业的问题。也许有些读者会说这是来自于Scott Adams 的"Seven years of highly defective people"一书(同样喷血推荐)。参见http://www.vim.org/iccf/click1.html的"re books and CDs"。

关于作者

Bram Moolenaar 是Vim的主要作者。他写了Vim的核心功能并且负责甄选其它作者的代码。他作为一名技术人员毕业于Delft技术大学，现在他主要从事软件业。但他也知道如何使用电烙铁。他是荷兰ICCF的创建者和出纳。这是一个帮助乌干达孤儿的组织。他作为一个系统建构师为自由软件工作，但实际上他为Vim花费了大量的心血。

用Vim进行C/C++编程介绍

作者:Kmj
slimzhao@hotmail.com 翻译整理

自从Bill Joy最初写出Vi编辑器以来，Vi就一直是编程者中广为流传的(即使不说是最流行的)编程工具.

Vi产生以来，历经不断革新，现在最新版的Vim已经具有了非常多的功能，这些功能使程序员能更加轻松，便捷地使用它们. 下面是它的一些功能描述，正是这些丰富强大的功能使vi和vim成为无数程序员的至爱. 本文志在向linux的初学者们介绍这些功能，而不是追溯其历史渊源. 对此感兴趣的读者可以查看"extra information"获得这些信息.

(注: 本文中使用vi兼指vim，但有一些选项可能只有vim支持)

ctags

Ctags是vim的伴生工具，它的基本功能是让程序员能自由穿梭于程序的不同部分(如从一个函数名跳转到该函数的定义处)，最通常的用法是象下面这样以源程序目录下所有文件作为参数.

shell command
```
[/home/someuser/src]$ ctags *
```

该命令会在当前目录下创建一个名为"tags"的文件，该文件包含了你当前目录下所有的C/C++文件中的相关信息，具体来说包含以下对象的信息:

List

- 由#define定义的宏
- 枚举值
- 函数定义，原型和声明.
- 类，枚举类型名，结构名和联合结构名
- 名字空间
- 类型定义
- 变量(定义和声明)
- 类,结构和联合结构的成员

接下来，Vim就通过该文件中的信息定位这些程序元素. 有几种方法可以对这些元素进行定位. 第一种方法，可以在命令行上启动vi程序时通过-t选项加要跳转的程序元素名，如下:

shell command
```
[/home/someuser/src]$ vi  -t  foo_bar
```

将会打开包含foo_bar定义的文件并定位到定义foo_bar的那一行上.

如果你已经在vi编辑环境中，也可以在底线命令行上键入:

ex command
```
:ta foo_bar
```

该命令可能使你离开你当前打开的文件(而跳转到包含foo_bar定义的文件的相关行上去，如果你已经改变了当前文件的内容而没有存盘，则只能在你设置了'autowrite'时才会跳转到该文件，否则会给出警告，另，autowrite可简写为等效的aw，译者注)，欲了解'autowrite'选项的详细信息，可以使用在线帮助:h autowrite 命令(也可简写为:h aw，译者注)

最后一种跳转到一个程序元素的方法是在(命令模式下)光标停在该程序元素上时按下CTRL-]键，如，你在看程序时看到某处调用了一个叫foo_bar()的程序，你可以将光标停在foo_bar单词上(停在该单词任何一个字符都可，译者注)，然后按下CTRL-]键，它就会跳转到该函数的定义处. 值得注意的是CTRL-]碰巧是telnet 的终端符，所以如果你在编辑远程计算机上的文件(通常是通过telnet登录到远程主机上，译者注)，可能会遇到一些问题. 通过在线帮助":h^]"可以了解这方面的更多信息[1].

Ctags程序也可用于其它语言写的源程序，并且可以与其它的一些编辑器(如emacs ，NEdit等等)协同工作. 正确地使用它，会给你的编程工作带来极大的便利，尤其是你在开发一个大的项目时.

关于ctags程序的更多用法，请参看它的相关帮助页，man ctags，或者通过vim的在线帮助系统查看它的用法，:h ctags

c语言风格的缩进

Vi有几种不同的方法实现自动缩进. 对于C/C++程序员来说，最好的方法显然是cindent模式，该模式具有多种功能帮助程序员美化程序的外观，无需任何额外的工作(当然，设置正确的模式:se cindent是必需的). 欲打开该模式，只需键入命令:set cindent(所有的set都可以简写为se，虽然只节省了一个字符，译者注) 需要注意的是cindent控制缩进量是通过shiftwidth选项的值，而不是通过tabstop 的值，shiftwidth的默认值是8(也就是说，一个缩进为8个空格，译者注)，要改变默认的设置，可以使用":set shiftwidth=x"命令，其中x是你希望一个缩进量代表的空格的数目.

cindent的默认设置选项一般来说是比较可人的，但如果你的程序有特殊需求，也可以改变它，设置cindent的选项，通过":set cino=string"选

[1]译者: 在:h]̂中关于该问题是这样说的，多数telnet都允许使用命令telnet -E hostname来打开或关闭该脱字符，或者用telnet -e escape hostname 来指定另外一个脱字符来代替]̂，此外，如果可能的话，可以使用rsh来代替telnet 来避免这个问题，关于telnet -E 及telnet -e的详情，请参看man telnet中相关的帮助

项(其中，string 是要用户自己键入的字符串，译者)，string定义了一个列表，该列表决定了你的cindent的行为．你可以定义多种indent类型，vim的帮助对此有很详细的说明． 欲查找关于该主题的帮助，使用命令":h cinoptions-values"． 要想查看当前的设置值，可以使用命令":set cino"．

要了解更多的细节，可以使用在线帮助":h shiftwidth"，":h cindent"，":h cinoptions"，":h cinoptions-values"，":h cinkeys"，和":h cinwords"

语法高亮

用过集成开发环境的程序员都知道语法高亮的妙处所在，它不光使你的代码更具可读性，它也使你免于拼写错误，使你明确注释的范围，Vim对多种语言都有语法高亮的功能，当然，C/C++一定包括在内，打开语法高亮功能，可使用命令":syntax on"． 如果你觉得默认的设置已经够好了，使用它就是如此简单． Vim的语法高亮工具也可以十分复杂，拥有众多选项．要了解更多的细节，可通过命令":h syntax"查看在线帮助，在支持彩色的终端上或者使用gvim(vim的GUI版，增强了一些功能，译者注)，但如果你当前的环境不支持彩色显示，vim会使用下划线，粗体字，试图进行等效的替代，但对我而言，这样太难看了．

要了解更详细的内容，可通过命令":h syn-gstart"，":h syntax-printing"查看在线帮助

编辑-编译-再编辑

这实在是极好的功能，其主要卖点是，你可以不用离开当前编辑环境，通过指定一个命令，就可以编译你当前编辑的项目，然后，如果编译时因发生错误而中断，vim将会打开第一个发生错误的文件并定位于引起错误的行上． 这一命令就是":mak" (或者":make")． vim将会运行由选项makeprg指定的make程序，它的默认值就是make． 如果愿意的话，你也可以使用命令":set makeprg =string"改变项目维护工具(比如，在VC下使用nmake，":set makeprg=nmake.exe" ，译者注)，

vim使用选项"errorformat"的设置去解析编译器输出的错误信息的格式． 由于不同的编译器有不同的错误信息格式，所以可能需要显式地指定错误信息的格式． 选项"errorformat"的设置使用与c函数scanf风格类似的语法，最重要的是指定%f，代表文件名，%l，行号，%m，错误信息．

GCC格式的errorformat设置:%f:%l:%m

有些编译器的errorformat可能十分复杂，但好在vim对此提供了完整的在线帮助":h errorformat"．

要了解其它细节，可用命令":h quickfix"，":h mak"，":h makeprg"，":h errorfile"，":h errorformat"查看相应的帮助．

有用的快捷按键

有一些快捷按键对程序员而言特别有用，下面是其中的一部分：

在函数中移动

List
```
[ [    = 移动到前一个行首的'{'字符上，等价于?^{
] ]    = 移动到下一个行首的'{'字符上，等价于/^{
[ ]    = 移动到前一个行首的'}'字符上，等价于?^}
] [    = 移动到下一个行首的'}'字符上，等价于?^}
{      = 到前一个空行上
}      = 到下一个空行上
gd     = 到当前局部变量的定义处(当前的意思是光标停留其上的单词).
*      = 到与当前单词相同的下一个单词上
#      = 到与当前单词相同的上一个单词上
''     = 到上次光标停靠的行
```

括号匹配:
% 可以让光标从它当前所在的括号跳转到与它相匹配的括号上去，对花括号和圆括号，方括号都有效，常用于手工检查括号是否匹对.

替换操作:
Vim具有强大的字符串替换功能，操作起来十分简单，不需惹人生厌的GUI(图形用户界面)，查找并替换文本，可以使用下面的命令:

ex command
```
:  [address] s//string/[g|c|N] (where N is an integer value).
```

(其中的N是一个整数值).

此命令查找由grep风格的正则表达式指定的匹配模式，并将其替换为由string指定的字符串，"address"，"g"，和"N" 是对命令的补充选项，它们分别决定了命令的作用范围，是只替换第一个匹配的字符串还是替换所有匹配的字符串，只替换每行中第N次匹配的字符串[1]:

List
```
g = 全部: 替换每行中所有匹配的字符串.
c = 询问: 在每次替换之前询问用户是否确定要进行替换.
N = Nth   只替换该行第N次匹配
        (不作指定时隐含为 N=1，替换该行的第一个匹配)
```

(即等价于address1，address2s//string/1)

[1]译者注: (a)如果没有指定这些辅助修饰标志，则vim默认为只替换一行中第一个匹配的字符串. (b) 据我所知，只有ed行编辑器才有这种品性，ex与vi都没有这个选项

List

```
[address values] --- 可以是一个或是由逗号分开的两个指定行范围的标识符.
(下面的x代表一个整数)
. = 表示当前行(即光标所在的行, 译者注)
$ = 当前文件的最后一行
% = 整个文件(即对每一行, 等价于1,$, 译者注)
x = 当前文件的第x行
+x = 从当前行开始下面的第x行(如果当前行为第1行, 则+3 代表第4行)
-x = 从当前行开始上面的第x行(如果当前行为第4行, 则-3 代表第1行)
```

逗号用于分隔任何上面指定的单个行，以形成一个范围(当然，这个范围的下界不能小于上界，如10,1为非法的范围，此时vim会给出一个警告信息，问你是否进行反向操作，如回答y，则等价于1,10，操作仍正常进行，否则，撤消当前操作，译者注)，其后指定的操作将作用于此处给出的范围，vim帮助里有关于替换操作的充分信息.

其它杂项

Vim有众多诱人的小功能，这里不可能一一列出，下面列出一些尤其值得注意的一些特性.

包含文件搜索---":h include-search"

书签设置---'mx'用于设置书签，''x'用于从书签返回;(其中的x可以为任何字母，但，只能记录当前文件里的书签) (退出vim后再次进入将不会保留这些书签,书签就是代表在文件中某一特定位置的一种标记，译者注)

"剪贴板" 缓冲- " ¨xY " 用于剪切或复制到一个名为x的缓冲区(Y代表任何的删除或取样命令)，' "xZ ' 用于粘贴内容(Z代表粘贴命令p或P)；(其中x可以为任何字母，也可在跳转到另一文件中时继续生效(:e filename).

注释符---":h comments"

.vimrc---别忘了你的.vimrc文件(在你用户目录中~/.vimrc). 该文件可用于记录上面你所做的大多数设置，记住在.vimrc文件中无需在每个命令前使用一个冒号":".(在DOS下的vim中，.vimrc文件放于vim程序所在的目录中，且，此时不叫.vimrc，叫_vimrc，另，.vimrc也可为.exrc，_vimrc也可为_exrc)

其它资源

X_Console(此处不知如何翻译，译者注)上有一个非常好的vi教程，如果你要开始学习使用vi，就从这里开始吧.因特网上有非常多的关

于vi/vim信息的网页，有好有坏(好坏也看你的水平如何了)在Google或其它搜索引擎上查找vi或vim会找到非常多的搜索结果，我个人觉得下面两个是最好的

VI爱好者主页---链接多多，信息多多...

VI帮助文件---非常完整而简练的一份参考手册，特别是ex命令.

Unix世界Vi教程---九部分，从开始到结束...看了就知道，我们为什么喜欢VI.

本文由Keith Jones(kmj9907@cs.rit.edu)所作；我不是vim专家，但我希望上面的一些内容对大家有所帮助. 希望大家喜欢!!!